PA039

Architektúra superpočítačov a intenzívne výpočty

Zápisky z prednášok

Superpočítače boli v minulosti špeciálne počítače vyrábané na zákazku (resp. vyrábané v malom počte), obvykle boli veľmi drahé. Od zvyšku počítačového sveta sa odlišovali špičkovými parametrami, ich cieľom bolo dosiahnuť maximálny výkon.
V súčastnosti (prakticky od začiatku 3. tisícročia) potrebuje dosahovať tak vysoký výkon čoraz viac skupín ľudí. V spojení s klesajúcou cenou hardware a jeho zvyšujúcim sa výkonom to malo za následok, že superpočítače sa podobajú štandardným počítačom s rozdielom, že v superpočítačoch je oveľa viac procesorov (napr. 18 000 procesorov x počet jadier = niekoľko miliónov jadier v jednom superpočítači). V dnešnej dobe sa tieto superpočítače používajú najmä na rozsiahle simulácie, modelovanie alebo predpovede počasia. Superpočítače v dnešnej dobe zvyšujú požiadavky na kvalitu programovania - programy musia byť schopné efektívne používať tisíce procesorov na počítanie jednej úlohy tak, aby čo najviac urýchlili výpočet.
Keďže procesory sú v dnešnej dobe oveľa rýchlejšie ako v minulosti, vznikajú nové požiadavky na HW a SW, aby dokázal s takým procesorom pracovať (napr. HW nedostatok - prístup do pamäte môže trvať niekoľko tisíc taktov procesora). V dôsledku šetrenia finančných zdrojov taktiež klesá pomer medzi teoretickým výkonom procesorov a dosiahnutým výkonom ⇒ nie je nutné mať superpočítač s extrémnym výkonom na nie až tak náročné výpočty
(v minulosti neplatilo, každý chcel mať top PC), naviac údržba extrémne výkonného superpočítača je drahšia ako údržba superpočítača menšieho.

High Throughput Computing
- snaží sa používať masívne výpočtové systémy tak, aby bolo dosiahnuté maximálne využitie celého systému, nie jeho najvyšší možný výkon
- namiesto jednej veľkej úlohy navrhuje počítať viacero menších úloh súbežne na viacerých procesoroch
- nie je podstatná rýchlosť spracovania úlohy, ale celkový čas spracovania všetkých úloh s maximalizáciou synchronizácie procesorov

Pamäte a procesory

Procesory

Parametre:

1. latencia - vyvolaná konečnou rýchlosťou prenosu signálu, prenosom signálu medzi procesorom a pamäťou a omeškaním v pamäti
2. rýchlosť obnovenia - rýchlosť prepínania obvodov a frekvencie prepínania. Pri zvyšovaní frekvencie prepínania nastáva problém s chladením obvodu.
3. priepustnosť - rýchlosť prenosu dát na čipe
4. granularita - koľko informácie je možné uložiť na jednotku plochy. Zhusťovanie pamäte ovplyvňuje latenciu.

Rozlišujú sa 2 druhy procesorov - CISC a RISC.

1. CISC Procesory

- Complete Instruction Set Computer
- v klasických počítačoch napr. PDP 11, VAX, IBM 370, Intel 80x86...
- z doby, kedy procesory neboli oveľa rýchlejšie ako pamäť → procesory si mohli dovoliť čítať dáta priamo z pamäte bez straty výkonu
- niekedy bolo čítanie z pamäte rýchlejšie ako samotný výpočet
- bolo výhodné, aby inštrukcie procesora boli čo najdlhšie
- časťou inštrukcie bola aj adresa operandu v pamäti (v súčastnosti sú operandy uložené v registroch procesora)
- do počítačov sa zvykli nahrávať rôzne sady inštrukcií. Napríklad, ráno sa na počítači počítali vedecké výpočty, tak sa tam nahrali inštrukcie vhodné na vedecké výpočty. Poobede sa šlo počítať len s celými číslami, tak sa počítač vypol, nahrali sa do ňho inštrukcie vhodné na počítanie s celými čislami a počitalo sa s celými číslami ... Počítače nemali v tej dobe dostatok pamäte na uloženie všetkých inštrukcií naraz.
- inštrukcie boli časom tak zložité, že nebolo možné urýchliť ich vykonanie - ani programátor nevedel ako veľmi je inštrukcia zložitá, pretože mohla zahŕňat veľké množstvo mikroinštrukcií
- rýchlosť výpočtu sa zvyšovala už len zvyšovaním taktu procesora
- paralelizácia v CISC procesoroch bola obtiažna - nutná analýza mikroinštrukcií

So vzrástajúcimi problémami s urýchľovaním výpočtov na CISC procesoroch sa zaviedol pojem **pipelining** - snaha využiť rôzne časti procesora na spracovanie inštrukcií v rovnakom čase.
Nech má každá inštrukcia 5 fáz, ktoré prebiehajú sekvenčne. Ak skončí prvá fáza prvej inštrukcie a začne sa spracuvávať druhá fáza, tak sa chce, aby sa zároveň paralelne spracúvavala prvá fáza druhej inštrukcie (nezrýchľuje sa vykonávanie inštrukcií, ale ich spracovanie). Ak v pôvodnom modeli bolo 5 inštrukcií, kde sa každá spracovala v 5 taktoch, spracovanie všetkých inštrukcií trvalo 25 taktov. Pomocou pipeline je možné rovnaké inštrukcie spracovať v 9 taktoch.
Pipelining rozoznáva 3 základné oblasti: spracovanie inštrukcií, prístupy k pamäti a výpočty v pohyblivej desatinnej čiarke.

Delenie inštrukcií na 5 fáz:

1. Instruction fetch - načítanie inštrukcie z pamäte
2. Instruction decode - rozoznanie/dekódovanie inštrukcie, tj. rozhodnutie, v ktorej vetve procesora sa má inštrukcia ďalej spracovať
3. Operand fetch - načítanie operandov - v CISC procesoroch to znamená ďalší prístup do pamäte
4. Execute - vykonanie inštrukcie
5. Writeback - zápis výsledku do pamäte

Pipelining v dnešných procesoroch:

1. neviditeľný pipelining - inštrukcie spracúva HW, avšak programátor (píše kód v strojovom jazyku) musí zvoliť vhodné poradie inštrukcií, aby zaistil efektívne využitie pipeline. Ak sa mu to nepodarí, HW zaistí správny výsledok, avšak negarantuje efektívne využitie.
2. viditeľný pipelining - je definovaný maximálny počet cyklov procesora na dokončenie výpočtu - inštrukcie musia byť písané v správnom poradí ⇒ programovanie v strojovom jazyku sa stáva veľmi náročným

2. RISC procesory

- Reduced Instruction Set Computer
- prvý v počítači CDC 6600
- vznikol ako dôsledok existencie konceptu pipeliningu, existencie problému rýchlosti procesora a rýchlosti pamäte (procesory začali byť rýchlejšie - CISC procesory to riešili zložitejšími inštrukciami, RISC zaviedol cache pamäť - vedie k vytvoreniu hierarchie pamätí) a existencie problému čítania zdrojových kódov pre CISC procesory. Dôsledkom je pokles ceny externej pamäte - bolo nutné ukladať dlhšie zdrojové kódy programov, pretože zanikli mikroinštrukcie (malé inštrukcie, pre programátora neviditeľné, ktoré vykonávali komplexnejšie inštrukcie), čiže bolo potrebné zapisovať viacero inštrukcií na vyššej úrovni ⇒ začalo byť náročnejšie programovať v strojovom jazyku, čo viedlo k vylepšeniu prekladačov - programátori už nepotrebovali vedieť assembler, stačil im vyšší programovací jazyk. Zápis programu do assembleru potom vykonal samotný prekladač.

Charakteristiky RISC procesorov:

- všetky inštrukcie sú rovnako dlhé ⇒ menej preklopení ⇒ vyšší výkon (pred tým to neplatilo, bolo treba instruction fetch)
- z CISC procesorov boli vybrané často používané a jednoducho (hardwarovo) implementovateľné inštrukcie (v pôvodných CISC procesoroch vykonávalo 20% inštrukcií 50% programu)(pôvodné RISC procesory nemali z toho dôvodu inštrukcie na výpočty s pohyblivou desatinnou čiarkou (považované za drahú operáciu) alebo nemali inštrukciu delenia (mali jednoduchšiu inštrukciu prevrátená hodnota))
- povedalo sa, že kvôli pipeline by malo vykonanie každej inštrukcie trvať zhruba rovnako dlho (po zavedení delenia sa to trocha rozbilo)
- všetky inštrukcie pracujú len s registrami. Jedinou výnimkou sú inštrukcie load a store na načítavanie alebo ukladanie dát, ktoré manipulovali priamo s pamäťou (priama adresácia, žiaden výpočet adresy)
- inštrukcia odložený skok - jedna z najčastejšie používaných inštrukcií v CISC bol JUMP, ktorý ale spôsoboval problém pre pipeline (menil poradie načítavania). Preto sa po inštrukcii JUMP načítali ešte ďalšie 2 inštrukcie, aby sa priestor v pipeline vyplnil.
- výrobcovia - HP, IBM (PowerPC), DEC
- ideál prvej generácie RISC procesorov: 1 takt procesora == 1 vykonaná inštrukcia

Výzkum v pipeline (doteraz):

- ako ušetriť energiu v procesore?
- pipeline zvyšuje spotrebu energie v procesore ale zároveň ho rozdeľuje do viacerých častí - otvára možnosť, že ak nie je potrebný plný výkon, je možné spomaliť pipeline (napr. plní sa len každý druhý takt) a tým častiam procesora, ktoré nič nerobia, sa nepošle energia ⇒ ak sa podarí efektívne riadiť príkon do častí procesora, zníži sa zahrievanie, čo by mohlo poskytnúť možnosť dosiahnutia ešte vyššieho počtu taktov proesora za sekundu.

Do RISC procesorov boli časom pridávané nové vlastnosti, ktoré ich mali zrýchliť. Vznikli:

1. superskalárne procesory
2. superpipelines
3. VLIW (Very Long Instruction Word)

Superskalárne procesory:

- mali viac ako jednu výkonnú jednotku: mali 2 aritmerické + 2 floating point (ďalej FP) jednotky, naviac mohli mať jednu špeciálnu jednotku na skoky a jednu pomocnú na aritmetické operácie (jednotky vedia pracovať paralelne)
- v dnešnej dobe maju takmer všetky procesory superskalárny charakter
- existuje paralelizmus vo vnútri HW - z pohľadu programátora je to obyčajný procesor, tj. programátor programuje sekvenčne, paralelizácia prebieha až vo vnútri procesora ⇒ komplikovanie procesora, musí obsahovať špeciálne obvody na rozhodovanie o paralelizme (paralelizovať sa nedá vždy, napr. P1 zapisuje do R1, P2 chce z R1 čítať, hovoríme že P2 je blokovaný procesom P1) ⇒ obvody musia byť schopné detekovať takéto prípady
- obsahujú inštrukciu MADD (multiply add) - zvládajú vykonať inštrukciu $x \cdot y + z$ v jednom kroku (narušuje koncepr RICS procesora, avšak bola zavedená, pretože používa 2 FP jednotky. V jednej jednotke sa vykoná $x \cdot y$, bez zápisu do registru druhá jednotka vezme výsledok a hneď k nemu pripočíta z). Bez tejto inštrukcie (s použitím registra na zápis medzivýsledku) by museli byť použité 2 inštrukcie, naviac pripočítavanie by bolo blokované.
- v superskalárnych procesoroch nemusia byť obe FP jednotky ekvivalentné. Jedna z nich napríklad nemusí vedieť deliť - delenie je drahá operácia, je na ňu potrebných viacero tranzistorov (predražuje sa výroba) + delenie nie je tak častá operácia, aby bolo treba 2 jednotky, čo to vedia.

Superpipeline:

- oproti základnej pipeline má viac fáz (napr. 25). Väčšina inštrukcií neprechádza všetkými fázami, pretože niektoré fáze sú špecifické len pre niektoré inštrukcie ⇒ dokáže paralelizovať nad rámec toho, ako to dokázala pipeline.

VLIW:

- princíp podobný ako superskalár - existuje viac jednotiek, ktoré je vhodné zamestnať
- nemajú žiadne HW kontrolné mechanizmy na rozhodovanie, či sú 2 inštrukcie na sebe závislé ⇒ zjednodušenie architektúry + zrýchlenie (nemusí rozhodovať)
- rozhodovanie prebieha na úrovni progr. jazyka alebo prekladača
- názov odvodený od princípu fungovania - namiesto x inštrukcií a n operandov má 1 inštrukciu, ktorá vykoná x inštrukcií a n operandov - HW ich vie spustiť naraz, pretože medzi nimi nie sú závislosti (resp. by mali byť vyriešené o úroveň vyššie). Väčsinou sa spracuváva viacero inštrukcií naraz, je to závislé od kvalitného prekladača (prinajhoršom bude 1 inštrukcia = 1 slovu procesora, takže oproti pôvodnému modelu to nespomalilo)
- dosahuje lepší výkon ako superskalár - prekladač môže pracovať s väčším kusom kódu (napr. s celou procedúrou) a preskladať ho tak, aby maximalizoval využitie jednotiek procesora ⇒ VLIW má oveľa lepšie využitie jednotiek procesora

Ďalšie rysy RISCov - obchádzanie, premenovanie registrov a skoky

1. skoky - sú problematické kvôli pipeline - čim viac jednotiek, tým sú skoky drahšie. Vznikli odložené skoky, tj. po inštrukcií skoku sa do pipeline nahrali ešte ďalšie 2 inštrukcie (kým sa vyhodnotilo, kam sa má skočiť). Riešenia:
   (a) nulovanie operácií - za inštrukciu skoku sa vkladajú inštrukcie, ktoré nič nerobia
   (b) pri podmienenom skoku - kým sa vyhodnotí podmienka, inštrukcie sa vykonávajú ďalej sekvenčne. Ak potom ku skoku naozaj došlo, vykonané inštrukcie bolo nutné vrátiť späť. Toto má zmysel, ak sa predpokladá, že ku skoku väčšinou nedôjde (vplyv na spotrebu energie, nie na rýchlosť výpočtu).
   (c) podmienené priradenie - pri podmienenom skoku sa (ešte pred vyhodnotením podmienky) začnú vykonávať inštrukcie z oboch vetiev, po vyhodnotení podmienky sa jedna vetva nuluje.
   (d) buffer potencionálnych cieľov skoku - v špeciálnej vyrovnávacej pamäti sa ukladajú predvypočítané výsledky, ktoré sa zatiaľ nepoužili (tj. výsledky z vetvy, ktoré sa pred tým zahodili, sa teraz uložia)
   (e) predpoveď cieľov skoku - dynamická (pri každom sa pamätá, ako sa podmienka vyhodnotila naposledy) a statická (fixované natvrdo, napr. vo for cykle). Počas vyhodnocovania podmienky sa potom predvypočítava vetva programu, ktorú procesor predpokladá.
2. premenovane/obchádzanie registrov - RISC procesory začinajú používať 2x toľko registrov, o koľkých vie prekladač - má možnosť lepšie optimalizovať na úrovni HW, používa paralelizmus (Z R1 číta, potom do R1 zapisuje, ak R1 medzi tým premenuje na R1', môže rovno zapisovať, ale pri ďalšom čitaní si musí pamätať, že má čitať z R1')

ANDES

- Architecture with Nonsequential Dynamic Execution Scheduling
- ultimate spôsob riešenia závislostí inštrukcií
- rozšírenie RISC o ďalšiu architektúru

- stále na úrovni superskalárov - v kóde jedna inštrukcia za druhou
- snaží sa rozšíriť tzv. okno inštrukcií, ktoré sa zvažujú pre paralelné spracovanie (ak je nasledujúca inštrukcia blokovaná, hľadá sa ďalšia, ktorá je vhodná)
- program sa potom vykonáva podľa množstva nezávislých inštrukcií v danom okne
- má viacnásobnú frontu inštrukcií - vezme inštrukciu, pozrie na príznak a na základe neho zaradí inštrukciu do vhodnej fronty
- 3 fronty - aritm. operácie, FP operácie, fronta na inštrukcie load/store
- problematická inštrukcia v RISC je napr ld r1 a (load do registru r1 z adresy a) - ak je pamäť pomalá, register r1 môže byť zaseknutý na stovky taktov procesora - obecne sa nevie riešiť na úrovni bežného RISC
- ANDES to zvláda, pretože na také inštrukcie má samostatnú frontu, funguje na princípe odložených load & store - sú 3 fronty inštrukcií, tie sa vyberajú z fronty na vykonanie podľa toho, či už majú pripravené dáta (paralelizácia presunu dát z hlavnej pamäte do registrov)
- aby to bolo možné, tak sa do pipeline pridáva tzv. "dokončenie inštrukcií"- časti inštrukcie fetch a decode sa vykonajú viditeľne navonok, časti issue, execute a complete sa vykonajú vo vnútri ANDES architektúry, graduate sa potom vykoná v poradí, v akom boli inštrukcie zapísané v pôvodnom kóde
- špekulatívny výpočet - ANDES predvypočítava rôzne varianty, avšak graduate vykoná iba pri niečom, ostatné zahadzuje (nutné dôsledné premenovanie registrov)
- veľkosť okna - 200 inštrukcií pri najvačšich ANDES architektúrach (berie do úvahy aj špekulatívne výpočty)
- inštrukcie skoku sa rozpoznávajú primárne
- využitie procesora podobné ako pri VLIW

Týmto je ukončený popis RISC procesorov a smeru, ktorým sa uberal ich vývoj. Dnešné počítače prestávajú ANDES používať. Po prvé, ANDES architektúra potrebuje zložitú logiku na rozhodovanie kto, kedy a kde ⇒ obsahuje veľa logiky na čipe, ktorá neprispieva k výpočtu. Po druhé, všetky architektonické prvky boli zamerané na maximalizáciu výkonu, nič iné sa vtedy do úvahy nebralo. V dnešnej dobe je ale snaha balancovať s výkonom aj spotrebu (ANDES arhitektúra má vyššiu spotrebu, ako by chceli zelení). Vyššia spotreba ⇒ vyšší výkon ⇒ viac tepla ⇒ obmedzenie možností zvyšovat takt procesora.

Dnešným trendom je púštať energiu len do tých častí procesora, v ktorých prebieha výpočet (obvody na riadenie prístupu energie sú jednoduchšie). Menšie zahrievanie potom umožní zvýšiť takt procesora, preto sa moderné architektúry vracajú k pôvodnému spôsobu - kompilátoru sa poskytne viac informácií o procesore na to, aby vedel rozumne kompilovať. Rozhodovanie efektívne/neefektívne sa necháva na prekladač. Na HW potom ostáva len kontrola veľmi jednoduchých závislostí, všetky zložitejšie by mal vyriešiť kompilátor (dnešné prekladače napr. vedia o 2x väčsom počte registrov, prípadne vie o ich premenovaní). V histórií sa múdre kompilátory považovali za veľmi zložité.

Pamäť

- jednoduchý pohľad - matica - má riadky a stĺpce (odvodené od starých feritových konštrukcií)
- adresa má potom 2 časti - index riadku a stĺpca
- page mode - čítanie matice (pamäte) po celých riadkoch (stránkach) > jedno slovo procesora (môže trvať rovnako dlho ako keby sa čítali len 2 políčka) ⇒ to, čo sa číta naraz z pamäte nie je organizačná jednotka, s ktorou pracuje program
- veľkosť bloku čítaného z pamäte odpovedá dvom parametrom - šírke zbernice, ktorou je pamäť napojená na procesor a veľkosti riadka vo vyrovnávacej pamäti. Cieľom je vedieť čítať a zapisovať celý riadok vo vyrovnávacej pamäti.

Parametre pamätí:

1. prístupová doba - čas, za ktorý procesor dostane dáta, o ktoré požiada (analogicky zápis) (v modeli matice to znamená vystavenie správneho riadka a stĺpca a prečítanie dát - niečo sa dá paralelizovať)
2. cyklus pamäte - ako dlho po tom, čo procesor niečo čítal, môže čítať niečo iné (čo spraví čítanie s pamäťou?) (niekedy je čítanie deštruktívne (dynamické pamäte) - po prečítaní je nutné ich znova zapísať, obvykle je blokovaný celý blok pamäti (submatica) kvôli zdieľaným obvodom)
3. možné zrýchlenie - nepracuje sa s jednou maticou ale s postupnosťou viacerých nezávislých matíc, tzn. čítanie dát z jednej matice neovplyvní čítanie z druhej matice. Úlohou je správne zorganizovať dáta v pamäti.

Virtuálne pamäte

- procesory pracujú s ešte jednou vrstvou pamäte - vedie k tomu, že všetky procesy, ktoré sa spustia, vidia len časť pamäte, s ktorou pracujú - pamäť sa tvári ako keby v nej nič iné nebolo
- všetky kusy pamäte pridelené procesom sú adresované od 1 - inštrukcie všetkých procesov pracujú s tým, že adresový priestor začína na jedničke ⇒ nutný preklad logickej adresy na fyzickú (zatiaľ ignorujeme stránkovanie)
- TLB - Translation Lookaside Buffer - mapovanie logickej adresy na fyzickú - transparentné pre proces, nie pre procesor (prepnutie kontextu vynúti prehranie adries v TLB, znovunačítanie hodnôt TLB znamená režiu ⇒ nedeterministické spomalenie výpočtu)
- TLB má konečnú veľkosť, pretečenie TLB znamená prehranie adries aj keď pracuje iba 1 proces
- v dnešnej dobe musia s TLB superpočítače počítať, pretože sú zložené s obyčajných procesorov (a tie pracujú s logickými adresami)

Vyrovnávacie pamäte

- medzi procesorom a vonkajšou pamäťou
- veľmi malá, veľmi rýchla, veľmi drahá
- cieľ - ak procesor potrebuje dáta, mali by sa nachádzať vo vyrovnávacej pamäti. Snahou je minimalizovať počet interakcií procesora s vonkajšou pamäťou
- vyrovnávacia a hlavná pamäť interagujú - interakcia by nemala ovplyvniť rýchlosť výpočtu (ako ANDES - neblokujúci load a store)

- hit pomer - ako často sa pri čítaní dát podarí nájsť dáta vo vyrovnávacej pamäti, tj. netreba siahať do vonkajšej (analogicky zápis)
- dnešok - 3 úrovne vyrovnávacej pamäte, prvé 2 úrovne zvyknú byť priamo na procesore, tretia úroveň má cca 64MB
- vyrovnávaciu pamäť je možné si predstaviť ako riadok fixnej dĺžky (128-256B), čo je viac ako veľkosť operandov, s ktorými procesor pracuje
- každý riadok by mal byť schopný niesť riadok z hlavnej pamäte (jeho priamu kópiu), avšak kapacita vyrovnávacej pamäte je menšia ako kapacita hlavnej pamäte ⇒ medzi pamäťami musí existovať mapovanie

Typy vyrovnávacej pamäte (z pohľadu mapovania na hlavnú pamäť):

1. priamo adresovateľná - o každom riadku vyrovnávacej pamäte vie, ktoré riadky hlavnej pamäte mapuje (napr. VP 4MB - obsahuje súvislý 4MB blok hlavnej pamäte) - lineárne usporiadané dáta zaplňujú celú vyrovnávaciu pamäť
2. množinovo asociatívná - množina riadkov z hlavnej pamäte sa mapuje do množiny riadkov vyrovnávacej pamäte, množiny sú primerane malé (najčastejšie používaná)
3. plne asociatívna - presný opak priamej - každý riadok hlavnej pamäte sa môže nachádzať kdekoľvek vo vyrovnávacej pamäti. Ideálne z pohľadu organizácie (do pamäte je možné kedykoľvek načítať ktorýkoľvek riadok v hlavnej pamäte), najzložitejšie z pohľadu logiky (práca s touto pamäťou stojí najviac energie a zaberá najviac priestoru na čipe)

Harward architektúra:
- v jednej časti pamäte sú dáta, v inej sú inštrukcie programov
- fyzicky je to jedna pamäť, logicky 2
- má oddelené aj vyrovnávacie pamäte (zvlášť dáta, zvlášť inštrukcie)
- 2 časti rôzne veľké, majú rôzne vlastnosti (typicky na inštrukcie plne asoc. pamäť, na dáta množinovo asoc. pamäť)
- z pohľadu programátora je plne transparentná - vidí ju ako celok

Ďalšie parametre pamäte:

1. Šírka toku dát
   - bandwidth - max priepustnosť pamäťového systému
   - v B/s - tj. koľko bytov za sekundu je možné čítať/zapisovať
   - je rôzna v rôznych častiach pamäte
   - najväčšia prepustnosť spravidla blízko pri procesore
2. omeškanie, latencia
   - doba medzi časom požiadavku a časom odpovede
   - rovnaká bez ohľadu na objem dát - snaha prenášať čím väčšie množstvo dát, kvôli minimalizácií latencie

Interleaved pamäť
- paralelizácia pamäte - rozdelenie hl. pamäte na menšie bloky, ktoré sa chovajú autonómne
- zvýšenie priepustnosti
- zníženie dopadu cyklu pamäte
- obvykle $2-8$ krát prekladané subsystémy - dáta musia byť chytro usporiadané, nie je možné pristupovať paralelne do jedného bloku dát
- môže zvýšiť latenciu, možnosť použiť pipeline na prístup do pamäte
- preskladanie prístupu k pamäti - ak systém vie, že má prekladanú pamäť, snaží sa garantovať, že load a store nejdú do rovnakého bloku, tzn. snaží sa dobre organizovať dáta (analógia ANDES pre pamäť, skôr experiment)

Najznámejšie procesory

Procesor MIPS R8000:
- zavedený v 1993 - plne 64 bitový - 4násobná superskalárna architektúra, 32+32 registrov pre celočíselné a fp operandy
- podmienené move inštrukcie
- fp jednotky postavené tak, že fungovali podobne ako vektorový procesor
- zvládol TB fyzickej pamäte
- 128b dátová zbernica
- TLVB dvojcestný, 384 položiek
- harward achritektúra - 16KB pamäte na inštrukcie (priamo adresovateľná, nepouživa TLB), 16KB na celočíselné operandy(priamo adresovateľná, 2 paralelné prístupy - 2 load alebo 1ld+1st paralelne, potrebuje TLB), 4MB streaming pamäť na výpočty s pohyblivou čiarkou
- extrémne rýchle fp výpočty - pre ne je procesor optimalizovaný (300 000 000 op/s, celočíselné výpočty 10x pomalšie)
- 2kB prediction cache - vyrovnávacia pamäť pre skoky, ukadá sa tam predikcia skoku

Procesor MIPS R10000:
- následník R10000
- snaži sa odstrániť nevyváženosť medzi veľmi rýchlou fp jednotkou a celočíselnou časťou
- superskalár, 4 inštrukcie paralelne
- plnohodnotná ANDES architektúra
- ak niekedy došlo k deleniu nulou, nastalo prerušenie (aj napr. v rámci špekulatívnych výpočtov napr.)
- R10000 sa vie prepnúť do módu s presným prerušením, tj. až v momente, kedy je taká inštrukcia graduovaná
- 32 registrov - ANDES $\Rightarrow$ fyzicky 64
- plné premenovanie pre fp aj celočíselné operandy
- plne asociatívny TLB
- rôzne dlhé stránky
- 128b datová zbernica, 40b adresová zbernica
- spoločná cache na celočíselné dáta, žiaden streaming
- vie predpovedať 4 cykly dopredu
- má neblokujúce inštrukcie ld a st
- má vyrovnávaciu pamäť druhej úrovne
- 2 ALU výpočtové jednotky, jedna špecializovaná na skoky, ďalšia na operácie násobenia a delenia
- 2 fp jednotky - nie sú rovnocenné - 1. je ščítačka a 2. je násobička - kvôli MADD
- fronty - celočíselná (16 položíek), floating (16 položiek), adresná

Procesor UltraSPARC-I:
- 1987
- prvý reálny 64b procesor, veľký adresný priestor - 64b fyzická a 64b virtuálna adresa - stránky rôznej veľkosti
- priamo integrovaná grafická jednotka - na spracovanie operácií pre chytrú grafiku
- na operácie s grafikou obsahuje špeciálne inštrukcie (GRU) - visual instruction set - sada inštrukcií na optimalizaciu grafických operácií priamo na procesore
- 16KB vyrovnávacie pamäte, blokujúci ld a st
- FPU - samostatná odmocnina a delenie, presné prerušenie
- GRU - 8 a 16b násobenie, 16 a 32b zhlukované sčítanie a booleovské inštrukcie, priamy prístup ku grafickej pamäti, priama podpora motion compensation

Procesor Intel Itanium (IA64):
- 64b architektúra, spätne nekompatibilná
- s 32 a 16b programami nevedel efektívne pracovať, bežali v tzv. emulačnom móde
- procesor nebol trhom prijatý (emulačný mód veľmi pomalý)
- ináč mal procesor veľmi dobre parametre
- druhá generácia navyšovala počet jadier

IMP Power7 procesor:
- pôvodne vyvíjaný pre superpočítače
- 8 jadier, 12 procesných jednotiek, 4 vlákna na 1 jadro
- L1 a L2 cache sú zvlášť na jadro, zdieľaná je až L3 cache mimo čip

IMP Power8 procesor:
- od 2004
- 8 vláken na jadro, 12 jadier
- rastie vyrovnávacia pamäť, L3 už na čipe
- nevýhoda je, že sú drahšie, - programy pre nich musia byť optimalizované a špeciálne preložené
- v dnešnej dobe patrí medzi najvýkonnejšie

Paralelné počítače

- prechod z intraprocesorového paralelizmu k procesorom s viac jadrami, odtiaľ potom k paralelným počitačom - viac procesorov spojených do vyšších celkov

Viacprocesorové systémy:
- zvyšovanie výkonu pomocou paralelizmu, už sa nedá zvyšovať takt procesorov
- explicitná paralelizácia - doteraz bol kód písaný pre 1 procesor, paralelizácia bola až na HW úrovni, tu sa paralelizuje už na úrovni zdrojového kódu
- vo viacprocesorovom systéme sa procesory musia spolu naučiť komunikovať - výmena dát cez hlavnú pamäť môže byť kritická, riešenia rôzne v závislosti od výrobcov
- počet procesorov, koľko vie výrobca spojiť: AMD 4, Intel 8, IBM 32

Delenie počítačov:

1. podľa počtu procesorov:
   (a) systémy **Small-scale multiprocessing** - 2-80 procesorov, procesory zdieľajú pamäť symetricky - všetci majú do pamäte rovnako ďaleko, rovnako rýchlo vedia čítať/zapisovať - je jedno ktorý procesor vykonáva operáciu
   (b) systémy **Large-scale multiprocessing** - stovky až tisíce procesorov, majú distribuovanú pamäť (tj. každý procesor má vlastnú časť) - vyššia doba prístupu do pamäte iného procesora.
2. podľa architektúry:
   (a) **SIMD** (Single Instruction Multiple Data) - odpovedá predstave o vektorovom počitaní, tj. všetky procesory synchronizované, vykonávajú rovnaké inštrukcie (dobré na prácu s maticami), nie sú vhodné na skalárne operácie. Procesory jednoduché, jednoduchší programovací model (ale zložité vlastné programovanie kvôli vektorovému počítaniu). Použitie ako GPU accelerators.
   (b) **MIMD** (Multiple Instruction Multiple Data) - predstaviteľné ako cluster so servrami. Plne asynchrónny model, samostatné procesory (netreba špecializovaný HW), veľká flexibilita - nie je nutné všetko vyjadrovať ako vektor, avšak zložitejšie programovacie modely (explicitná synchronizácia v kóde).
3. podľa programovacích modelov
   (a) **SPMD** (Single Program Multiple Data) - Celému paralelnému počítaču sa zverí jedná úloha, tá sa vo vnútri stroja rozloží na podúlohy, ktoré počítač rieši paralelne, môžu mať vektorový aj skalárny charakter.
   (b) **MPMD** (Multiple Programs Multiple Data) - paralelný počítač v daný moment počíta viacero nezávislých programov, výsledky programov sa na konci skombinujú a prezentujú (vhodné pre MIMD).

Vektorový procesor:
- primárne pracuje s vektorom dát
- má vektor ako dátový typ inštrukčnej sady a má špec. inštrukcie na prácu s vektormi
- špecifikum - vektorový load a store, nenačítavajú dáta z pamäte lineárne, ale vedia pracovať s indexmi, tzn. vedia do jedného vektoru načítať dáta z rôznych častí pamäte

- ak je pamäťový systém postavený ako interleaved, tak indexy sú v ideálnom prípade v rôznych blokoch pamäte ⇒ procesor vie požiadavky na vektorový load a store veľmi efektívne paralelizovať
- vektorové slovo je obvykle oveľa dlhšie ako počet procesorov (napr. 1024 slov a 16 procesorov - spracovávajú slova po 16ticiach)

Komunikačné modely

1. Zdieľaná pamäť - vyskytuje sa zväčša v malých systémoch, pamäť je oddelená od procesorov, najjednoduchšie je pripojiť pamäť zbernicou, so zvyšujúcim sa počtom procesorov klesá výkon (pamäť je pomalá, prenosová kapacita zbernice je obmedzená). Zložité prekladanie výpočtu a komunikácie (aktívne čakanie).

2. Odovzdávanie správ - vyskytuje sa zväčša v distribuovaných systémoch, každý procesor má vlastnú pamäť (iné nevidí, ale vie o nich, existencia adresácie). Ak chcú procesory medzi sebou komunikovať, musia dáta zabaliť a odoslať ⇒ vysoká cena komunikácie (žiadosť, zabalenie dát, vytvorenie správy, odoslanie, prevzatie správy, rozbalenie). Výhodou je jednoduchšie prekladanie výpočtov - 1 vlákno čaká, iné môžu počítať niečo iné.

Hybridné systémy

1. NUMA - každý procesor má vlastnú pamäť, pamäte sú prepojené zbernicou, tzn. procesory vidia aj pamäte iných procesorov (neuniformná pamäťová architektúra, pamäte sú rôzne vzdialené). Prístup do cudzej pamäte môže byť veľmi drahý, vysoká škálovateľnosť, v prípade použitia vyrovnávacej pamäte je nutná synchronizácia pamätí.

2. COMA - experimentálny model, ak chce procesor čítať dáta zo vzdialenej pamäte, dáta sa prekopírujú do nejakej pamäte, ktorá je bližšia. Vytvára akúsi hierarchiu "vyrovnávacích"pamätí, nutná synchronizácia, zložitá logika.

3. DSM - každý procesor má svoju pamäť, ale programátor vidí na SW úrovni len jednu. HW riešenie - posielanie správ medzi procesormi sa vykonáva transparentne, SW riešenie - programátor musí program prispôsobiť.

Koherencia vyrovnávacích pamätí
- pričiny výpadkov vyrovnávacej pamäte:

1. Compulsory miss - procesor chce dáta, ktoré nie sú vo vyrovnávacej pamäti

2. Capacity miss - nízka kapacita vyrovnávacej pamäte

3. Coherence - vo vyrovnávacej pamäti je stará kópia dát

4. Conflict - rôzne adresy mapované do rovnakého miesta

Riešenie problému koherencie
- broadcoastová vyrovnávacia pamäť - pamäť kt. používa zbernicu (tzv. **snoopy cache**) a sleduje komunikáciu po zbernici (zápis dát do hl. pamäte) a pozerá, či sa nejdú prepisovať dáta, ktorých kópia je vo vyrovnávacej pamäti - reakcia:

1. **zneplatnenie** - dáta vo vyrovnávacej pamäti sú prehlásené za neplatné, v prípade opätovného prístupu sú dáta znova nahrané z hlavnej pamäte

2. **aktualizácia** - dáta sa hneď aktualizujú (čítajú sa priamo zo zbernice).

- existuje problém falošného zdieľania, tj. 2 procesory pracujú nad rovnakým riadkom, ale nad inými časťami riadku.
- univerzálne sa nedá rozhodnúť, či je lepšie zneplatnenie alebo update
- nevýhoda snoopy cache - nepoužiteľné pri veľkých systémoch, pretože by museli byť všetky procesory pripojené na jednu zbernicu

Možné riešenia:
- **adresárový prístup** - v adresári sa pamätá, ktoré riadky hlavnej pamäte majú kópiu v nejakej vyrovnávacej pamäti. Zaujímavé sú len kópie na čítanie (kópia na zápis je exkluzívna).
- **použitie vektoru príznakov** - pre každý procesor a každý riadok v pamäti sa drží príznak, či má procesor vlastnú kópiu dát (n procesorov $\Rightarrow$ n+1 bitový vektor (1 bit na exkluzívny zápis)). Problém nastáva pri zvýšení počtu procesorov, pretože je nutné rozšíriť dĺžku vektora - pamäť na vektor nemusí stačiť a dokupovanie dodatočnej pamäte je veľmi drahé. Naviac, takmer všetky vektory sú nulové (počet procesorov $\ll$ počet riadkov v pamäti). Ďalej sa nestáva často, aby veľa procesorov čítalo jeden riadok v pamäti.
Preto začali vznikať trocha chytrejšie prístupy (ako cena sa uvažuje počet abstraktných krokov, ktoré musí systém vykonať na synchronizáciu pamätí):

1. **Plne mapované adresáre**
   - používané, ak chce nejaký procesor zapisovať, tj. požiada o exkluzívny prístup k riadku v pamäti
   - zapíše sa exkluzívny bit a jednotka ako príznak, že s daným riadkom procesor pracuje
   - všetkým ostatným procesorom, ktoré s riadkom pracovali, sa pošle správa o zneplatnení riadka
   - exkluzívna 1 je dôležitá, aby mohli byť všetky požiadavky na vytvorenie počas zápisu pozdržané

2. **Čiastočne mapované adresáre**
   - namiesto bitového vektoru sú použité ukazatele, ktoré ukazujú na procesory s kópiou dát vo vyrovnávacej pamäti
   - existuje pool ukazateľov na procesory
   - ukazatele v poole je možné reťaziť v prípade, že k jednému riadku pristupuje viacero procesorov

3. **Obmedznené adresáre**
   - je obmedzený počet procesorov, ktoré môžu paralelne pristupovať k jednému riadku
   - riadky si môžu "vypožičiavať" voľné sloty (v prípade dynamickej alokácie)
   - môže nastať pretečenie - všetkým procesorom sa môžu zneplatniť dáta a príde nová množina procesorov, alebo sú dáta zneplatnené len jednému procesoru. Ďalšia možnosť je použiť na adresovanie strom (tzv. Coarse-vector) - v prípade pretečenia sa výpočet posunie o úroveň vyššie, kde bude jeden ukazateľ odkazovať na skupinu procesorov (broadcast na zneplatnenie sa potom presúva len na daný podstrom)

4. **Previazané adresáre**
   - doteraz sa info o tom, kto kde má dáta, držali vedľa hlavnej pamäte, ale držidelia dát sú vyrovnávacie pamäte
   - v previazaných adresároch - ak procesor žiada o dáta, tak sa info o tom (ukazateľ) uloží do hlavnej pamäte. Ak bude chcieť nejaký další procesor čítať rovnaký riadok, tak sa na ňho nebude odkazovať z hlavnej pamäte, ale z procesora, ktorý už kópiu má (nový sa zaradzuje na začiatok zoznamu)
   - výhody - v hl. pamäti práve jeden odkaz (+ príznak exkluzivity) nezávisle na počte procesorov, rovnako jeden odkaz v pamäti procesorov
   - vyššia škálovateľnosť
   - negatíva - použitie lineárneho zoznamu, zneplatnenie/update trvá dlhšiu dobu (tj. drahší zápis)

5. **Hierarchické adresáre**
   - hierarchické zbernice, snoopy broadcoast, adresáre

Zapájanie procesorov do sietí

Vlastnosti siete:

1. **Rozšíriteľnosť (škálovateľnosť)**:
   - ideál - s rastúcim počtom procesorov klesá doba spracovania úlohy
   - systém je rozšíriteľný, ak toto (skoro) platí (nemusí platiť donekonečna, v skutočnosti sa to ani nedá)
   - dôležitý pomer $\frac{\text{cena za rozšírenie}}{\text{cena za zrýchlenie}}$
   (napr. pridanie 3 procesorov je ešte ok, pridanie štvrtého môže vyžadovať zmenu logiky, lepšiu dosku, atď.)

2. **Zrýchlenie**
   - ako sa zmení rýchlosť výpočtu úlohy na 1 procesore oproti výpočtu na n procesoroch
   - ideál - po pridaní n procesorov sa výpočet n krát skráti a zároveň sa n krát zjednoduší komunikácia
   - realita - výpočet nie je n krat kratší, pretože je potrebné doplniť nejakú logiku, tj. v niektorej časti výpočtu sa musi použiť viacero inštrukcií
   - tiež je problém s kapacitou zdieľanej zbernice - zrýchlenie taktiež závisí na schopnosti paralelizácie riešenia problému
   - **Amdalov zákon** - paralelizovateľnosť problému ovplyvní maximálne zrýchlenie spôsobené pridaním procesorov - riešenie sériového problému sa nezrýchli pridaním procesorov

Rozšíriteľné prepojovacie siete
- požiadavky na ideálnu sieť - nízka cena rastúca lineárne s počtom procesorov (N), latencia nezávislá na N, priepustnosť rastúca lineárne s N

Tri základné komponenty sietí:

1. Topológia
2. Prepínanie dát
3. Smerovanie

Parametre siete:
- počet uzlov N
- stupeň uzlu d
- polomer siete D
- redundancia siete A - koľko hrán je potrebné odstrániť, aby sa sieť rozdelila na dve siete
- cena - počet liniek v sieti
- Bisection width Bw - šírka rozpolenia - min. počet liniek, ktoré je potrebné odstrániť, aby sa sieť rozdelila na 2 rovnaké siete. (čím viac, tým lepšie - vyššia priepustnosť medzi 2 polovicami grafu)
- Bisection bandwidth Bb - celková kapacita odstránených liniek v bisection width - ideál - $\frac{\text{bisection bandwidth}}{N}$ konštantný

Topológie sietí

1. **Jednorozmerné prepojovacie siete**
   - najjednoduchšie s najhoršími vlastnosťami
   - procesory usporiadané jeden za druhým, spojené jednou linkou - tzv. korálky
   - nízka cena
   - má zmysel do 4 procesorov
2. **Dvojrozmerné prepojovacie siete**
   - kruh - pridanie jednej linky k jednorozmernej sieti - stupeň uzlov rovnaký, polovičný polomer oproti korálkam, bisection width 2, o jednu linku viac ako korálky, má zmysel do 8 uzlov
   - hviezda - dobrý polomer, vysoká záťaž pre uzol v strede, redundancia 1, Bb určená stredom, dobrá cena $n-1$ liniek
   - stromy - často používané - dobrý polomer siete, zlá priepustnosť (problémový koreň), redundancia 1, n-árny strom má maximálny stupeň uzlu $n+1$
   - tučný strom - pridanie linky medzi uzlami s každou úrovňou stromu, z koreňa bude najviac liniek - zlepší Bw
   - dvojrozmerná mriežka - najčastejšie používané, dobrý polomer, redundancia 2, vyššia cena, Bw je $\sqrt{N}$
   - torus - uzavretá mriežka - všetky uzly stupeň 4,výrazné zníženie polomeru oprotu mriežke ($\sqrt{N}$), zvýšenie Bw na $2 \cdot \sqrt{N}$, zlá škálovateľnosť - nové uzly nutné pridavať po riadkoch/stĺpcoch
3. **Hyperkocka**
   - viacrozmerná sieť
   - veľmi dobré základné parametre - polomer $\log n$, $Bw = \frac{N}{n}$, redundancia $\log n$, vyššia cena
   - binárne číslovanie uzlov - ľahké nájdenie cesty
   - typické pre dnešné paralelné počítače (možné v kombinácií s torusom)
4. **Plne prepojené siete**
   - teoretické
   - každý uzol má toľko spojov, koľko tam je ďalších uzlov
   - všetky parametre super, neakceptovateľná cena

Prepínanie
- konkrétny mechanizmus, ako sa paket (správa) dostane zo zdrojového uzlu (procesora) do cieľového

1. **Prepínanie paketov (store-and-forward)**
   - celý paket sa uloží na uzle
   - na uzle je buffer, keď sa celý zaplní, dáta sa pošlú ďalej
   - periodická kontrola integrity dát - často ale zbytočná a dlhá
   - vysoká latencia $(P/B) * D$ - dĺžka správy P, priepustnosť B, počet hopov D

2. **Prepínanie okruhov**
    - 3 fáze zahájenie spojenia - probe, vlastný prenos, zrušenie spojenia
    - latencia $\frac{P}{B} \cdot D + \frac{M}{B}$ - P (omnoho menšie ako pred tým) dĺžka vzorku, M dĺžka správy (P<<M)
3. **Virtuálne prepojenie (cut-through)**
    - snaha odstrániť počiatočnú latenciu pri vytvorení cesty
    - správa rozdelená na menšie bloky - flow control digits (flits)
    - prvý flits obsahuje info o ceste, ďalšie flitsy obsahujú dáta, posledný flits ruší cestu
    - flitsy posielané kontinuálne - zaplnenie kapacity linky (v prechádzajúcich prípadoch to nebolo vynútené)
    - latencia $\frac{HF}{B} \cdot D + \frac{M}{B}$ - ako pred tým, HF je dĺžka flitsu
4. **Smerovanie červou dierou (wormhole routing)**
    - flits je schválne veľký ako buffer v uzle
    - buffre maju rovnakú veľkosť
    - podporuje replikáciu paketov ⇒ vhodné pre multicast a broadcast, bez straty rýchlosti
5. **Virtuálne kanály**
    - zdieľanie fyzických kanálov
    - do uzlu idú 2 spoje, po oboch sa prenáša plnou prenosovou kapacitou, uzol je preťažený ⇒ pre každý spoj vlastný buffer, odosielateľovi treba oznámiť, nech posiela nižšou rýchlosťou
    - využitie - preťaženie spojenia, zábrana deadlocku, mapovanie logickej topológie na fyzickú, garancia priepustnosti pre systémové dáta

Smerovanie v prepojovacích sietiach:

- hľadanie cesty:

1. statické (napr. hyperkocka) - odosielateľ presne vie, kadiaľ pôjdu dáta, vytvárajú sa preťažené linky
2. adaptívne - dynamické, berie do úvahy zaťaženie siete

Fault tolerance:
- dlho sa predpokladalo, že linky sú bezchybné
- teraz sa často používajú samoopravné kódy
- opakované posielanie správ a potvrdzovanie správ sa nepoužíva - veľmi by to zaťažilo systém

Omeškanie pamäte:
- pamäť pomalšia ako procesor
- riešenia:

1. zrýchlenie prístupu
    (a) **NUMA** - Non Uniform Memory Access - niektoré pamäte sú od niektorých procesorov viac vzdialené ako iné, je snaha aby dáta boli v pamäti blízko procesoru, zložitejšie programovanie

(b) **COMA** - experimentálne, možnosť presúvania riadkov pamäte, optimalizovaná práca s kópiami dát

2. prekryv prístupu a výpočtu

(a) **Modely slabej konzistencie**
- paralelný výpočet je rozdelený do blokov, synchronizácia prebieha medzi blokmi výpočtu - odpadá nutnosť stáleho synrchonizovania hodnôt premenných v pamäti.
- operácia acquire = zablokovanie premennej procesorom, iné procesory nevidia
- operácia release = uvoľnenie premennej procesorom - garancia, že hodnota premennej nemusí byť aktuálna, hodnoty sa synchronizujú len pri fence (tj. release na všetky premenné), kým sa všetky nesynchronizujú, musia všetky čakať

(b) **Prefetch**
- presun dát do vyrovnávacej pamäte s predstihom (ako v ANDES architektúre)
- nonbinding prefetch - negarantuje existenciu požadovaných dát vo vyrovnávacej pamäti (záleži aj na iných vláknach)
- binding prefetch - zaisťuje presun dát až k procesoru, možnosť narušenia konsistencie
- existuje prefetch na HW aj SW úrovni
- prefetch-exclusive - žiadosť o dáta s tým, že procesor do nich bude môcť zapisovať, žiadosť o exkluzivitu

(c) **Procesory s viacnásobným kontextom**
- procesor je schopný súčastne spracuvávať viac vláken, 1 vlákno je spracuvávané kým druhé čaká na dáta z pamäte (prepnutie kontextu trvá 1-2 takty procesora)

(d) **Komunikácia iniciovaná producentom** - podobné NUMA - producent produkuje dáta pre konzumenta, zapisuje ich do pamäte v blízkosti konzumenta
- vhodné pre presun veľkých blokov dát

Podpora synchronizácie

- Základné synchronizačné primitíva:

1. **Vzájomné vylúčenie**

- zaisťuje, že v danom momente má prístup k premennej len jedno vlákno
- nutná HW podpora - 2 inštrukcie: test&set a test-and-test&set
- test&set - všetky vlákna neustále kontrolujú zámok a čakajú, kým sa otvorí (tzv. busy waiting), zamknutie zámku musí byť atomická operácia (zaisťuje HW), kontrola otvoreného zámku môže byť vykonaná v danom momente len jedným vláknom
- test-and-test&set - vlákno najprv kontroluje, či je zámok otvorený a ak je otvorený, tak ho znova otestuje a až tak zamyká = citlivejšie ako test&set
- nevýhoda je, že všetky vlákna, čo čakajú, sú zamestnané, je možné predbiehanie (protokol negarantuje kto pôjde prvý)
- použitie fronty - vlákna sú usporiadané vo fronte, zaistí poradie prístupu k zámku (prebúdzanie vláken)
- zámky v multiproceosoroch - existencia inštrukcie, ktorá vykoná viacero inštrukcií atomicky (napr. prečítanie a zvýšenie hodnoty premennej alebo compare&swap)

2. **Dynamicke rozloženie záťaže**

3. **Informácia o udalostiach** - namiesto synchronizácie cez pamäť sa zasiela signál (udalosť), nie je potrebný busy waiting (použitie v prostredí producenta a konzumenta)

4. **Globálna serializácia** - bariéry

Paralelizmus - Prekladače

Optimalizujúci prekladač:
- uvažujeme prekladač do medzijazyka
- fáze optimalizácie - source to source (napr. z javy do javy, ale ten druhý kód je "lepší"), preklad do medzijazyka, preklad do strojového kódu, optimalizácia behu (berie do úvahy precíznu znalosť procesora)(napr. v ANDES)

preklad do medzijazyka, oblasti:
- medziprocedurálna analýza
- optimalizácia cyklov
- globálna optimalizacia

medzijazyk:
- obsahuje štvorice (n-tice) - operátor, 2 operandy, výsledok
- obsahuje skoky
- obsahuje pamäť

Bloky:
- pri optimalizácií sa vytvorí graf toku
- blok - časť programu bez skokov (1 vstupný a 1 výstupný bod)
- 1 blok = orientovaný acyklický graf
- optimalizácia vo vnútri bloku = odstránenie opakovaných podvýrazov, odstránenie prebytočných premenných

Klasické optimalizácie:

1. Propagácia kopírovaním - odstránenie závislostí na poradí vyhodnotenia inštrukcií, umožnenie paralelizmu

2. Spracovanie konštant - propagácia konštant (v kóde sa 2+3 prevedie na 5), problematické je delenie nulou alebo veľké čísla, strata presnosti (tvorcovia prekladačov by na problémy mali myslieť a ošetriť)

3. Odstránenie mŕtveho kódu - šetrenie cache pamäte

4. Strength reduction - vhodné v prípade, že niektoré inštrukcie sú pomalšie ako iné (napr. $2 \cdot k$ sa prevedie na $k + k$)

5. Premenovanie premenných - ak sa do premennej uloží medzvivýsledok a potom sa tá ista premenná použije ako iný medzvivýsledok - vytvorí 2 samostatné premenné na 2 medzivýsledky - umožní paralelizmus

6. Odstránenie spoločných podvýrazov - počítanie adresy prvku v poli - výpočet prebieha vzhľadom k začiatku poľa

7. Priradenie registrov premenným - pri preklade do medzijazyka sa neuvažuje reálny počet registrov na procesore (je ich tam neobmedzene), táto skupina optimalizácií sa snaži nájsť vhodné mapovanie premenných na reálne registre procesora

8. Odstraňovanie smetia - použitie inliningu namiesto volania funkcie, reorganizácia podmienených výrazov, zrušenie nadbytočných testov (napr. tautológie), zrušenie vyhodnotenia podmienky v cykle v prípade, že sa výsledok testu nezmení v každej iterácií (môže urobiť programátor), rozbalenie cyklu (problém s obmedzenou presnosťou desatinných miest), odstránenie konverzie typov (kvôli kontrole presnosti)(ukážky v slajdoch compiler2015.pdf)

Optimalizácie cyklov

1. Redukcia réžie

2. Zlepšenie prístupu k pamäti

3. Zvýšenie paralelizmu

Dátové závislosti:
Flow dependency - aktuálny výpočet závisí na predchádzajúcom výsledku
Anti dependency - zložitejšia závislosť v cykloch
Output dependency - niekedy je kód zložitý a počítajú sa tam hodnoty, ktoré sa nepoužijú.

Optimalizácia cyklov

- loop unrolling - zníženie množstva iterácií cyklu tak, že sa v jednej iterácií spočíta to, čo sa pred tým spočítalo vo viacerých iteráciách

- zníženie réžie

- zvýšenie paralelizácie v ramci jedného procesora (viac ALU)

- loop unrolling nutné adaptovať na skutočný počet iterácií (napr. pri znížení počtu iterácií na štvrtinu, musí byť pôvodný počet iteracií deliteľný štyrmi). Zvyšné iterácie sa buď vynechajú, spočítajú pred cyklom (pre-conditioning loop) alebo po cykle (post-conditioning loop)

- nevhodné cykly - malý počet iterácií, cykly s volaním procedúr, cykly s podmienenými výrazmi, veľké cykly

- problémy - rozvoj so zlým počtom iterácií, zahltenie registrov, výpadky vyrovnávacích pamätí, hw problémy (cache coherence, preťaženie zbernice)

- optimalizácia prístupu k pamäti - optimalizácia prístupu k prvkom dvojrozmerného poľa, obrátenie indexov (niektoré jazyky ukladajú matice po riadkoch, niektoré po stĺpcoch). Pri práci s maticami je tiež možné pracovať len s časťou matice (blokom), ktorý sa vojde celý do cache.

PVM

Parallel Virtual Machine:
- motivácia - tvorba paralelného virtuálneho superpočítača
- postupné prepojovanie pracovných staníc
- intuitívne - virtuálnosť tohoto stroja spočíva v tom, že programátor nemá jeden superpočítač, ale má veľa obyčajných počítačov, ktoré spolu vedia komunikovať
- distribuované prostredie pre vývoj a spúšťanie distribuovaných programov
- virtuálny stroj tvorený kooperujúcimi úlohami - samostatné procesy, ktoré bežia na rôznych procesoroch
- prirodzená podpora pre task paralelizmus (Multiple Programs Multiple Data)

komponenty:

1. sieť pvmd démonov - procesov, ktoré vytvoria virtuálny počítač, na každom PC v clustri beží démon
2. knižnica funkcií - API pre medziprocesovú komunikáciu, manipuláciu s procesmi, dostupná pre C, C++ a Fortran

Základné triedy príkazov v knižnici

1. Riadenie procesov - každý proces ma identifikátor (TID - task identifier), nesie informácie o úlohe a o jej umiestnení v PVM, existuje jeden riadiaci démon na správu procesov (ak tento démon spadne, spadne celá PVM). Základné príkazy: pvm_mytid, pvm_exit, pvm_kill
2. Posielanie a prijímanie správ - odovzdávanie správ sprostredkované démonmi, sú zodpovedné za spoľahlivý prenos a doručenie, správu vždy preberá lokálny pvmd, následne z TID cieľoveho procesu určí jeho umiestnenie a správu pošle vzdialenému pvmd procesu. Zasielanie správ musí byť možné medzi rôznymi architektúrami. Posielanie správ môže byť blokujúce aj neblokujúce.
3. Správa buffrov - odosielanie správ realizované prostredníctvom buffru - správa sa nakopíruje do buffru a jeho obsah sa potom posiela.
4. Skupinove operácie - PVM podporuje tvorbu skupín procesov - procesy je možné priradzovať a odoberať
5. Informácie
6. Signály
7. Pokročilé funkcie - umožňujú nahradiť defaultné správanie v komunikačnom modele nejakým iným

PVM - primárne zamerané na budovanie paralelného stroja, málo funkcií na zasielanie správ
MPI - menej možností na správu procesov, viac funkcií na zasielanie správ, dôraz na prácu s dátmi, paralelné I/O

MPI

- Message Passing Interface
- komunikačné rozhranie pre paralelné programy
- definované API - je štandardizované, existujú nezávislé implementácie (napr. optimalizované pre konkrétny HW)
- príklady iplmementácií - SGI MPI, OpenMPI
- sada funkcií, ktoré umožňujú vytvoriť distribuovaný program
- určené pre SPMD/MIMD
- optimalizujúci prekladač - nemôže optimalizovať ako chce, mohol by zmeniť sémantiku programu

Vývoj špecifikácie MPI

- MPI 1.0 - nebola nikdy implementovaná, väzba na jazyky C/Fortran
- MPI 1.1 - oprava nedostatkov, už bola implementovaná
- MPI 1.2 - prechodová verzia pred MPI 2.0
- MPI 2.0 - obsahovalo rozšírenia pre paralelné IO operácie, manipuláciu s procesmi, má väzbu už aj na C++, obsahuje operácie get/put (vzdialený zásah do pamäte iného procesu bez toho, aby o tom vedel)
- MPI 3.0 - remote memory access, fault tolerance, ...

Ciele MPI

- prenositeľnosť - definícia štandardu nezávislého na implementácií a jazyku
- výkon - optimalizácie pre HW
- funkcionalita - snaha pokryť všetky aspekty medziprocesorovej komunikácie

Jadro funkcionality MPI

1. MPI_Init - inicializácia MPI - vytvorenie prostredia
2. MPI_Comm_Size(MPI_COMM_WORLD, &size) - zistenie počtu procesov (počet procesov, ktoré zdieľajú komunikátor MPI_COMM_WORLD)
3. MPI_Comm_Rank(MPI_COMM_WORLD, &rank) - zistenie vlastného identifikátoru
4. MPI_Send(buffer, len, dest, tag) - zaslanie správy
5. MPI_Recv(buffer, maxlen, source, tah, actlen) - prijatie správy
6. MPI_Finalize - ukončenie MPI - bariéra všetkých procesov, vynucuje prijatie všetkých odoslaných a zatiaľ neprijatých správ

Klasický prístup posielania správ

- odosielateľ vykoná send a príjemca receive - dáta zaslané ako prúd bytov - príjemca musí správne dekódovať - práca programátora
- nutné poznať identifikátor príjemcu
- synchronizácia (by default je to synchrónna komunikácia)

- problém buffru - dáta musia byť v buffri ešte pred odoslaním, musí to strážiť programátor, problém častého kopírovania z/do buffru (príjem/odoslanie)
- tag správy - definuje poradie prijatia správ, sú globálne, komplikácia pri použití viacero nezávislých knižníc, príjemca môže špecifikovať aký príznak očakáva - kolektívne operácie vyžadujú mnoho príkazov send a receive (neefektívne)

Rozšírenia MPI
- procesy môžu byť usporiadané do skupín
- každá správa má nejaký kontext - zaslanie a prijatie správy je možné len v rámci jedného kontextu
- skupina a kontext spolu definujú komunikátor. (MPI_COMM_WORLD - skupina tvorená všetkými procesmi MPI programu)

Dátové typy:
- typ v MPI je popísaný trojicou (adresa, počet, datový typ) - názov typu prefixovaný s MPI (napr. MPI_INT)
- existujú zložitejšie typy - polia MPI typov, matice $\cdots$
- krokované pole MPI typov - rieši problém usporiadania matice v pamäti, má 2 časti - dĺžku riadka matice (segment) a index stĺpca ktorý chce programátor sprístupniť (offset)
- neexistujú ukazatele - v distr. prostredí nemajú význam
- umožnuje vytvorenie vlastných typov

Point to point komunikácia:
- odoslanie správy medzi 2 procesmi

1. blokujúce (synchrónne) - ďalší výpočet možný až po receive, čaká sa, realizovaný pomocou handleru a flagu
2. neblokujúce (asynchrónne) - opak blokujúceho
3. bez buffru - vždy blokujúce, odosielateľ nemôže prepisovať príslušnú časť pamäti skôr, než si to príjemca prečíta
4. s buffrom:
   (a) blokujúce - správa sa pošle, výpočet pokračuje po nakopírovaní správy do buffru
   (b) neblokujúce - výpočet pokračuje hneď, môže prebiehať paralelne s kopírovaním dát do buffru

- prijatie správy medzi 2 procesmi

1. blokujúce - čaká sa na dokončenie prijatia správy
2. neblokujúce - na prijatie správy sa vytvorí nové vlákno, výpočet prebieha ďalej

- buffer môže byť spravovaný MPI (častejšie) alebo programátorom
- existuje mnoho ďalších kombinácií
- odosielateľ môže stále zistiť, či príjemca už prijal správu
- možné vytvoriť persistentné kanály

Kolektívne operácie

1. broadcast - MPI_BCAST - odoslanie správy všetkým procesom
2. redukcia - MPI_reduce - opak broadcastu, vyžaduje prijatie správ od všetkých ostatných procesov (napr. agregácia medzivýsledkov zo všetkých procesov do jedného procesu)

Virtuálne topológie
- umožňujú vytvoriť SW topológiu nezávisle na skutočnom zapojení procesorov
- zvyšujú efektivitu pri písaní programu
- zvyšujú prenositeľnosť programov - programy nie sú závislé na reálnej topológií
- možnosť optimalizácie implementácie pre rôzne reálne topológie
- napr. matica, kruh, torus, hyperkocka...

MPI umožnuje vykonávať operácie so súbormi:
- paralelne, volá sa to parallel IO
- podpora od MPI 2.0
- vhodné pri ukladaní medzivýsledkov do zdieľaného súboru, každý proces vidí len svoju časť súboru (každý proces má svoj offset)
- ukladanie dát do súboru je neblokujúce

Profiling a Benchmarking

- profiling - meranie času behu konkrétnych častí programu, obvykle pred/po optimalizácií
- benchmarking - porovnanie systémov
- unixový prikaz time:

1. user time - čas procesu stráveného na procesore mimo kernel
2. system time - čas procesoru strávený obsluhou finkcií jadra
3. elapsed time - celkový čas výpočtu,

- CPU time = user time + system time
- vysoký podiel systémového času môže znamenať, že sa príliš často volajú funkcie jadra
- ak je CPU time výrazne menší ako čas výpočtu, znamená, že program dlho čaká na nejakú udalosť

Profilovanie
- profil obecne - množina dát (obvykle v grafickej forme), ktorá zachytáva významné vlastnosti niečoho (teraz programu)
- poskytuje informácie, pomocou ktorých môže programátor neskôr optimalizovať - zaujímavé je dynamické chovanie programu
- vypovedá o interakcií programu a systému, na ktorom program beží
- profilovanie prebieha pomocou SW nástrojov
- pri profilovaní sa zbiera najmä - graf volania funkcií, dáta o výkone pamäte, udalosti súvisiace s architektúrou (napr. predikcia skokov), sada informácií, ktoré ponúka priamo operačný systém
- program z pohľadu profileru - dynamická postupnosť vykonávania blokov

Typy profilerov

1. **Call graph profiler** - ukazuje časy a frekvencie použitia funkcií alebo procedúr, môžu byť statické alebo dynamické (detailnejšie, ale profil zložitejší na analýzu), spomaľujú beh programu, pretože za behu musia generovať a ukladať dáta o jeho behu
2. **Event based profiler** - obvykle pri interpretovaných jazykoch, profilovanie vykonáva runtime, tiež spomaľujú výpočet

3. **Statistical profiler** - periodicky zastavuje beh programu a vtedy sa pozerá na nejaké čítače. Ide o prerušenia na úrovni OS, spomaľuje beh výpočtu menej ako 2 predchadzajúce, nevýhodou je nepresnosť - je nutné mať dobre nastavený interval vzorkovania. Vôbec nepotrebuje prístup k zdrojovému kódu, stačí skompilovaná binárka.

Inštrumentovanie programu - vloženie inštrukcií do programu, ktoré umožnia vytvorenie profilu (napr. nejaké výpisy), je nutné mať pristup k zdrojovému kódu. Môže byť manuálne (programátorom), vykonané prekladačom alebo za behu.
Stopa výpočtu - surové dáta, ktoré sa profilerom zapíšu behom výpočtu. Potom sa na analýzu zbierajú stopa výpočtu, informácie o výpadkoch pamäte, stránky, hodnoty čítačov, atď. Tieto dáta potom slúžia na identifikáciu úzkych miest vo výpočte, ktoré sú potom adeptmi na optimalizáciu. Pred analýzou je nutné vedieť, čo sa chce optimalizovať (rýchlosť výpočtu, spotreba energie, atď.) a podľa toho potom profilovanie nastaviť. Taktiež sa na profilovanie dá pozerať dvomi spôsobmi, a to že je pripravený HW a cieľom je na ňom vyladiť program (obvyklé), alebo je daný program a otázkou je, čo je nutné urobiť s HW, aby na ňom tento program bežal lepšie (napr. pri návrhu nového HW). V dnešnej dobe je moderná optimalizácia spotreby energie (napr. zmenou veľkosti pipeline), často za cenu spomalenia výpočtu.

Benchmarking
- snahou je porovnanie systémov, cieľom je na základe referenčných dát zistiť, ktorý HW je lepší pre daný program(tzn. neumožnuje spustiť program na všetkých možných architektúrach, resp. spustenie programu má na starosti druhá strana, ktorá dodá nejaký výsledok)
- nie je možné vykonať absolútne usporiadanie

Základné prístupy:

1. súkromné benchmarky - je daný program, ten sa spustí na viacerých HW (to nie je v réžií toho, kto vyvýja program). Program spúšťajú viaceré organizacie (výrobcovia HW) na svojich strojoch, tie potom vývojárom programu vrátia nejaké výsledky, na základe ktorých sa potom vyberie cieľový stroj (vyberie sa dodávateľ stroja, na ktorom programátor chce, aby program v budúcnosti bežal).

2. priemyslové benchmarky - cieľom je zistiť všeobecné chovanie systému, nie pre beh špecifického programu

Porovnanie výkonu počítačov, benchmarky:

- MIPS - milión celočíselných inštrukcií za sekundu

- MFLOPS - milión operácií s pohyblivou desatinnou čiarkou za sekundu

- MIPS a MFLOPS o ničom nevypovedá (ostatné vlastnosti môžu byť veľmi zlé)

- VAX MIPS - program, ktorý za sekundu výpočtu na počítači VAX 11780 vykoná 1 milión operácií. Potom sa program spustí na inom PC a zisťuje sa pomer počtu vykonaných operácií.

- Linpack - knižnica pre aritmetiku, počítajú sa operácie s maticami. Meria sa rýchlosť vynásobenia 2 matíc a zisťuje sa, pri ako veľkých maticiach je systém najrýchlejší. Používa sa na rebríček prvých 500 najrýchlejšich PC.

SPEC benchmarky:
- organizácia Standard Performance Evaluation Corporation
- predchádzajúce benchmarky boli veľmi špecifické, neodpovedali širokému použitiu počítačov
- vytvorila sa množina programov (tzv. kernel kódy), pomocou ktorých sa testujú vlastnosti systému
- tieto programy sa potom testujú v SPECu

SPEC skupiny benchmarkov:

- OSG - Open System Group
- HPG - High Performance Group
- GPC - Graphic Performance Characterization Group

- nové kernely OSG - šach, hra go, kompresia videa, kvantové výpočty, spracovanie XML, hľadanie najkratších ciest (GPS), ...

Transakčné benchmarky - na porovnanie výkonu databází (používajú napr. banky)
Sieťové benchmarky - kapacita siete (programy netperf, iperf), priepustnosť, ...
Vlastné benchmarky - pre špecifické požiadavky

Kontrola benchmarku:
- nutné overiť, či sa meralo to, čo sa merať chcelo
- je nutné minimalizovať ovplyvnenie merania okolitými podmienkami (napr. pri testovaní 3 procesorov je nutné použiť vždy rovnakú pamäť, resp. spúšťať to vždy na rovnakom OS, atď.)
- nutná explicitná kontrola CPU času, výsledkov programu (musia byť správne) a porovnania hodnoty benchmarku so známym štandardom

CUDA a GPU

Moorov zákon - počet tranzistorov na jednom PC sa každých 18 mesiacov zdvojnásobí
- skôr - zlepšovala sa schopnosť spracovať 1 programové vlákno
- dnes - paralelizmus
- úlohovy paralelizmus - dekompozícia problému na úlohy, program popisuje úlohy, vhodný menší počet výkonných jadier
- dátovy paralelizmus - hľadá paralelizmus v spracovaní dátovych štruktúr (sčítanie vektorov) - z pohľadu vývoja procesora je to jednoduché, pretože všetky ALJ robia to isté, procesory majú vyššiu dátovú priepustnosť
- grafické výpočty sú dátovo paralelné

GPU
- GPU procesory sú výkonnejšie ako CPU
- GPU sú lacné, dajú sa na nich písať aj koncové aplikácie
- pužitie GPU - chemické výpočty, simulácie, fyzika v hrách...

Architektúra GPU
- viac paralelné ako CPU - desiatky multiprocesorov
- in order (GPU neprehadzuje poradie spracovania inštrukcií)
- SIMT - programátor píše skalárny kód, GPU berie množinu vláken, na ktoré aplikuje jednu inštrukciu

(CPU sú MIMD a SIMD)
- GPU má menšiu CACHE
- GPU používa viac tranzistorov ako CPU

HighEndGPU
- sú zarazené do zbernice
- koprocesor s dedikovanou pamäťou
- asynchrónne inštrukcie
- prepojenie CPU s GPU cez PCI-E (PCI Express)

Procesor G80
- prvý procesor architektúry CUDA
- 16 multiprocesorov, každý z nich má 8 skalárnych procesorov, 2 jednotky pre špeciálne funkcie (špeciálne jednotky na urýchlenie výpočtov s nižšou presnosťou na časté grafické výpočty - napr. goniometrické funkcie), 768 vláken - ak požadované dáta nie sú v pamäti, tak sa spustia iné vlákna (maskovanie latencie pamäti funguje pomocou prepínania vláken na HW úrovni)
- 8192 registrov
- 16Kb zdieľanej pamäte pre vlákna na 1 multiprocesore
- zvlášť pamäť pre textúry, na čítanie, vidia ju všetky vlákna zo všetkých multiprocesorov

Porovnanie rýchlosti GPU a CPU
- teoreticky - GPU cca 10x rýchlejšia aritmetika, 5x vyššia priepustnosť pamäte
- prakticky - výkonnostný rozdiel môže byť vyšší (pre špecializované operácie) aj nižší (nevhodná paralelizácia)
- vhodné problémy pre GPU - sčítanie veľkých vektorov, redukcia (suma prvkov jedného vektoru), problém nájdenia povodňovej mapy (2D mapa, každý bod v nejakej výške, z 1 bodu tečie voda, počíta sa, kam sa voda dostane (úlohový paralelizmus))
- nevhodné problémy pre GPU - ak rôzne vlákna vykonávajú rôzne činnosti (divergentný kód) alebo dáta sú nevhodne rozmiestnené v pamäti (divergencia prístupu do pamäte, napr. usporiadanie vrcholov grafu v pamäti, riedke matice)

Latencia GPU
- niektoré výpočty nemá zmysel na GPU počítať (aj keď su dobre paralelizovateľné) kvôli PCI-E zbernici, pretože je pomalá, tj. presun dát z procesora do GPU trvá dlhú dobu
- aby malo zmysel dáta na GPU dáta prenášať, je nutné, aby GPU vykonalo dostatočne veľký počet aritmetických operácií (sčítanie vs. násobenie matíc)

CUDA
- Compute Unified Device Architecture
- architektúra pre paralelné výpočty
- umožnúje efektívnu implementáciu výpočtov na GPU
- najpoužívanejší jazyk - C for CUDA
- nutné oddeliť CPU a GPU kód
- hierarchia vláken:
- 2 úrovne - vlákna sú organizované do blokov, bloky sú organizované do mriežky. Vlákna v jednom bloku majú garantované, že bežia na jednom multiprocesore (rýchla synchronizácia)
- pamäte majú rôznu viditeľnosť, rozdielne rýchlosti, rozdielnu životnosť dát v pamäti

Hierarchia pamätí:
- registre - najrýchlejšie, obsahujú lokálne premenné a medzivýsledky, majú životnosť vlákna
- lokálna pamäť - obsahuje to, čo sa nevojde do registrov, uložená v DRAM, vyššia latencia, životnosť vlákna
- zdieľaná pamäť - pomalšia ako registre, dynamická veľkosť, životnosť bloku
- globálna pamäť - najpomalšia, latencia v stovkách GPU cyklov, životnosť aplikácie, novšie su cacheované

- barierová synchronizácia - v CUDE default, musia do nej vstúpiť všetky vlákna
- atomické operácie - read-modify-write operácie v zdieľanej alebo globálnej pamäti - realizované pom. atomických premenných
- synchronizácia pamäťových operácií - ak sa chce programátor uistiť, že ukladané dáta sú viditeľné pre ostatné vlákna
- synchronizácia medzi blokmi - slabá podpora v CUDE (nemôžu čakať všetky vlákna vo všetkých multiprocesoroch)

Optimalizácia pre GPU

Prístup do globálnej pamäte
- tá je všeobecne najpomalšia, je potrebné k nej pristupovať rozumne
- delená do 64B segmentov, tie sú po dvoch združené do 128B segmentov
- GPU prenáša z hlavnej pamäte slová dlhé 32B, 64B alebo 128B - snaha maximalizovať veľkosť prenášaného slova
- nezarovnaný prístup - íte vlákno čita i+1. element, potom i+2, i+4...
- prekladaný prístup - vlákno číta prvky, ktoré sú od seba veľmi vzdialené - výkon klesá viac ako pri nezarovnanom
- zavedenie L1 a L2 CACHE, L1 pre multiprocesor, L2 zdieľaná medzi všetkými vláknami GPU
- overfetching - ak sa cachuje viac pamäti, ako je žiadané, napr. z globálnej pamäte sa žiadajú 4B, ale musí sa CACHEovať 128B, pretože taká je veľkosť riadku CACHE

Zdieľaná pamäť
- 16 alebo 32 pamäťových bank, k jednotlivým bankám sa chce pristupovať paralelne
- broadcast - ak chce viacero vláken čítať rovnakú hodnotu
- konflikt bank - 2 rôzne vlákna z jedného multiprocesoru chcú pristúpiť k rôznym bankám - prístup sa serializuje

Ostatné pamäte - ďalšie spomalenia
- prenosy medzi systémovou a grafickou pamäťou - nutné minimalizovať, výhodné prenášať veľké kusy naraz (vypnutie stránkovania), výhodné prekrývať výpočet s prenosom
- texturová pamäť - ako CACHE
- pamäť konštant - optimalizovaná na broadcast
- registre - možná read-write latencia (2. inšt. čaká na výsledok 1.)

Transpozícia matíc
- výstup - problém prekladaného prístupu (napr. zápis 4B čísla s použitím 32B transakcie)
- odstránenie prekladania - matica sa rozbije na dlaždice, tie sa lokálne transponujú, nakoniec sa transponujú dlaždice
- veľkosť dlaždice - riadok deliteľný 16
- ďalší problém - bank konflikt - pri čítaní globálnej pamäte sa zapisuje do zdieľanej po riadkoch, pri

zápise do globálnej pamäte sa číta zo zdieľanej pamäte po stĺpcoch - čítanie s prekladaním
- riešenie - zväčšenie naalokovanej pamäte v zdieľanej pamäti, aby sa nepracovalo s násobkom veľkosti riadku

- pomalé inštrukcie na GPU - celočíselné delenie
- rýchlejšie inštrukcie - sin, cos... (ale so znížením presnosti)
- cykly - veľký overhead - skoky
- update riadiacej premennej - riešenie pomocou tzv. unrollingu - kompilátor vezme telo cyklu a nakopíruje ho niekoľko krát pod seba

Súčet prvkov vektorov
- problém podtečenia - ak sa k strašne veľkému floating point číslu pripočíta strašne malé floating point číslo ⇒ z veľkého čísla je vidno len konečný počet desatinných miest, takže tá malá hodnota ho vôbec nezmení. To spôsobuje problém, ak sa k veľkému číslu pripočítava veľké množstvo malých čísel
- paralelný algoritmus - ščítava prvky vektora po dvojiciach, potom sčítava po dvojiciach výsledky, atď.
- v takom algoritme je potrebné mať globálnu bariéru
- nepekné - pre sčítanie n čísel je potrebných $2n$ medzivýsledkov
- využitie rýchlejšej pamäte - v jednom bloku sa neberú dvojice, ale m-tice prvkov - výhodnejšie z hľadiska prístupu do pamäte
- stále vysoká úroveň divergencie - klesá počet pracujúcich vláken
- riešenie - preusporiadanie indexov prvkov tak, aby boli nepracujúce vlákna v jednom warpe - ale to spôsobí konflikty bank
- riešenie 2 - snaha pristupovať k súvislej časti pamäte súvislým kusom vláken - prerobenie indexov tak, že í-te vlákno pripočítava í-ty prvok z prvej polky s í-tym prvkom z druhej polky - zefektívnenie práce s pamäťou
- časť vláken stále nič nerobí, kým sa neuvoľní blok - môže sa vykonávať časť sčítania uz počas načítania dát z pamäte
- posledný pracujúci warp sa unrolluje - ručne (pomocou nejakej šablóny sa to vie urobiť počas kompilácie)
- cena paralelizmu - určenie počtu vláken/procesorov tak, aby sa s minimálnym počtom vláken nestratil výkon (resp. viac vláken neurýchli výpočet)

Vypracoval R. Wittner, 396340, jar 2015