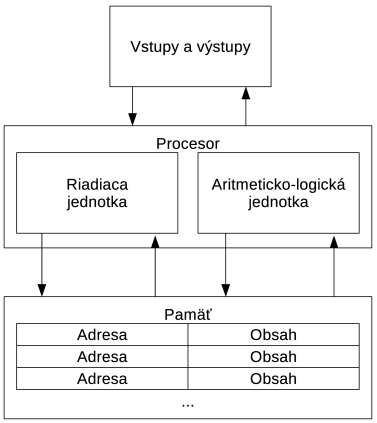
Hardvér pre paralelné počítanie

V mnohých vedných oblastiach, obzvlášť v prírodovedných a technicky orientovaných vedách, je bežnou praxou vytváranie a používanie paralelných programov. K tomu je však nevyhnutné poznať hadvérovú architektúru a softvérové prostriedky. Len tak je možné navrhnúť a napísať efektívne pracujúci paralelný program. V tejto kapitole si objasníme niektoré skutočnosti z historického vývoja, ktoré zohrávajú významnú úlohu pri dosahovaní čo najvyššej efektivity programu a oboznámime sa s rôznymi druhmi paralelných architektúr. Zároveň sa pokúsime zhrnúť nevyhnutné poznatky pre správne rozhodovanie sa pre výber vhodnej architektúry a efektívne implementovanie paralelných programov.

1 Historický vývoj

Našim hlavným cieľom je oboznámiť sa hlavne s modernými počítačovými architektúrami umožňujúcimi spúšťanie paralelných programov. Všetky tieto moderné architektúry však majú pôvod v sériových architektúrach, ktoré umožňovali beh iba jednej úlohy v tom istom čase. Z toho dôvodu si najprv objasníme niekoľko základných poznatkov z tejto oblasti.

Koncept digitálneho počítača, ktorý má program uložený v pamäti bol navrhnutý v roku 1936 Alanom Turingom a po prvýkrát bol takýto stroj s názvom EDVAC (Electronic Discrete Variable Automatic Computer) zostrojený v roku 1949 Johnom Mauchlym a J. Presper Eckertom [1, 2]. Základná koncepcia väčšiny bežne používaných počítačov je postavená na princípoch architektúry Johna von Neumanna, ktorý v roku 1946 spolu s kolegami špecifikoval, ako by mal vyzerať a fungovať počítač [3]. Základnými prvkami, ktoré by mal obsahovať, sú hlavná pamäť – RAM, ktorá obsahuje miesta označené adresami, na ktorých sa môžu uchovávať buď inštrukcie alebo údaje. Ďalej obsahuje procesor (CPU) s aritmeticko-logickou (ALJ) a riadiacou jednotkou (RJ) a radič pre vstupno-výstupné operácie. Táto koncepcia je viac-menej s rôznymi modifikáciami a vylepšeniami zachovaná do súčasnosti tak, ako je znázornená na obrázku 1.



Obr. 1: Schéma von Neumannovej architektúry počítača

Podľa tejto koncepcie procesor pozostáva z dvoch hlavných častí. Riadiaca jednotka je zodpovedná za určenie inštrukcie, ktorá sa má vykonať a aritmeticko-logická je zodpovedná za samotné vykonanie tejto inštrukcie. Nevyhnutnosťou pre vykonávanie inštrukcií sú dostupné údaje uložené vo veľmi rýchlej pamäti procesora, ktorú označujeme ako registre. Riadiaca jednotka pracuje so špeciálnym typom registra nazývaným počítadlo programu, ktorý obsahuje adresu nasledujúcej inštrukcie, ktorá sa má vykonať.

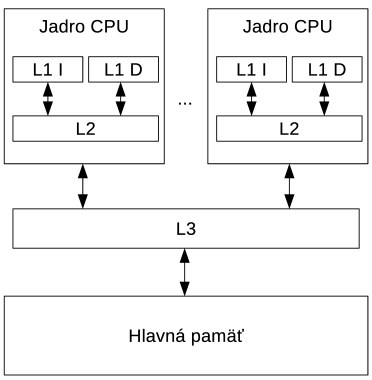
Jedným z hlavných nedostatkov tejto koncepcie je potreba prenosu údajov medzi registrami procesora a hlavnou pamäťou. Podľa tejto

koncepcie počítač vykonáva v rovnakom čase iba jednu inštrukciu nad malou množinou údajov. Tieto je najprv potrebné načítať z hlavnej pamäte do registrov a po vykonaní inštrukcie zas zapísať výsledok z registrov do hlavnej pamäte. Prepojenie umožňujúce tento prenos však spravidla je oveľa pomalšie, ako je rýchlosť vykonávania inštrukcií, čo spôsobuje výrazné spomalenie behu programu. Súčasné procesory sú schopné vykonávať inštrukciu viac ako 100-krát rýchlejšie, než je možné načítať údaje z hlavnej pamäti do registrov. Ako riešenie problémov tejto koncepcie bolo zavedených niekoľko modifikácií v podobe použitia rýchlej vyrovnávacej pamäte – cache, virtuálnej pamäte a inštrukčného paralelizmu [4–6].

Rýchla vyrovnávacia pamäť

Pri vykonávaní programu potrebuje procesor načítavať z hlavnej pamäte nasledujúcu inštrukciu alebo dáta, s ktorými je potrebné inštrukciu vykonať. Kým sú tieto načítané cez zbernicu a pripravené v registroch, procesor musí čakať. Aby sa skrátil tento čas čakania, bol zavedený akýsi medzistupeň medzi procesorom a hlavnou pamäťou v podobe vyrovnávacej pamäte. Vyrovnávacia pamäť – cache predstavuje relatívne malú pamäť s podstatne rýchlejšou dobou prístupu v porovnaní s hlavnou pamäťou. V tomto prípade sa budeme zaoberať cache pamäťou CPU. Táto spravidla býva umiestnená na tom istom čipe ako CPU alebo na inom čipe, ku ktorému má CPU rýchly prístup [5, 7].

Hlavná funkcia cache pamäte spočíva v tom, aby boli potrebné dáta rýchlejšie pripravené pre spracovanie procesorom a aby bol výsledok po spracovaní rýchlejšie uložený. Otázkou však je, ktoré dáta a inštrukcie by mali byť v cache pamäti pripravené, alebo ktoré dáta a inštrukcie bude procesor v najbližšej dobe požadovať. Pre vyriešenie tohto problému sa predpokladá, že pri vykonávaní programu sú inštrukcie vykonávané bezprostredne za sebou a potrebné dáta uložené v pamäti blízko za sebou, pretože po vykonaní jednej inštrukcie sa spravidla vykonáva nasledujúca inštrukcia a pri spracovaní údajov uložených v poli sú prvky tohto poľa uložené v pamäti jeden za druhým.



Obr. 2: Schéma členenia úrovní cache pamäte CPU

Použitie rýchlej vyrovnávacej pamäte – cache umožňuje zefektívniť komunikáciu procesora a hlavnej pamäte po zbernici tak, že namiesto prenosu jednej inštrukcie alebo jedného údaju sú po zbernici prenášané naraz celé bloky obsahujúce inštrukcie alebo údaje. Zároveň je takto možné použiť zbernicu s väčšou šírkou, aby bolo možné blok preniesť čo najrýchlejšie. V prípade, že CPU požaduje nejakú informáciu z pamäte, predpokladá sa, že v blízkej dobe bude požadovať aj informácie uložené v jej bezprostrednej blízkosti, a preto je do cache pamäte načítaný celý blok – riadok. Riadok cache pamäte spravidla pozostáva z 8 až 16 hodnôt. Po načítaní do cache pamäte je možné čítať tieto ďalšie údaje oveľa rýchlejšie ako z hlavnej pamäte. V prípade, že sa požadovaný údaj v cache pamäti nenachádza, je potrebné načítať z hlavnej pamäte do cache pamäte ďalší blok.

V súčasnosti je bežnou praxou, že procesory disponujú viacerými úrovňami cache pamäte, ako je znázornené na obrázku 2. Dôvodom je fakt, že cache pamäť, ktorá je najbližšie k jadru CPU, a teda je najrýchlejšie dostupná, má malú kapacitu. Označuje sa ako L1 cache a pozostáva z dvoch častí a to L1 inštrukčná cache a L1 dátová cache, ktorých kapacita je v súčasnosti rádovo niekoľko desiatok kB. Druhá úroveň cache – L2 cache je pomalšia ako L1 cache a má kapacitu rádovo niekoľko stoviek kB. Väčšina dnešných bežne používaných procesorov disponuje aj treťou úrovňou cache pamäte L3 s kapacitou rádovo niekoľko MB. V takejto hierarchii je najprv požadovaná informácia hľadaná v L1 cache pamäti, v prípade nenájdenia sa postupuje na L2 cache pamäť a následne na L3 cache pamäť.

Na nasledujúcom príklade programu 1 si prakticky ukážeme funkciu cache pamäte pri sčítaní dvoch veľkých matíc. Matice sú dvojrozmerné polia, kde ku každému z prvkov pristupujeme pomocou dvoch identifikátorov – indexov: index riadku a index stĺpca. Keďže pamäť počítača je len lineárna sekvencia postupne adresovateľných buniek, neumožňuje priamo uložiť štruktúru pozostávajúcu z riadkov a stĺpcov. Pre uloženie takéhoto dvojrozmerného poľa v pamäti sa v jazyku C používa spôsob ukladania po riadkoch. To znamená, že v pamäti je najprv uložený prvý riadok, za ním druhý, tretí, atď., čiže celá matica je vlastne uložená ako jedno dlhé jednorozmerné pole.

Poznámka: Niektoré jazyky, napríklad Fortran, používajú formát ukladania matíc po stĺpcoch.

Poznámka: Dynamicky alokované dvojrozmerné pole v jazyku C ako pointer na pointer nemusí mať riadky matice uložené v pamäti bezprostredne za sebou.

Úloha: Zmerajte rýchlosť programu pre sčítanie dvoch matíc podľa uvedeného zdrojového kódu a programu, v ktorom bude vymenené poradie cyklov na riadku 4 a 5 tak, aby program prechádzal prvky matíc po stĺpcoch.

Zdrojový kód 1: Kód pre sčítanie dvoch veľkých matíc

```
1  double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
2  int i, j;
3
4  for(i = 0; i < MAX; i++) {    //cyklus pre index riadku
5    for(j = 0; j < MAX; j++) {  //cyklus pre index stĺpca
6      C[i][j] = A[i][j] + B[i][j];
7    }
8  }
```

Keďže sú prvky matíc spracovávané v tom istom poradí, ako sú uložené v hlavnej pamäti, náš program bude plne benefitovať z funkcie cache pamäte.

Rozhodovanie o tom, ktoré údaje budú uložené v cache pamäti nie je v réžii programátora. Preto je potrebné, aby si programátor bol vedomý týchto skutočností pri návrhu a implementácii programu, aby boli dáta spracovávané a inštrukcie vykonávané podľa možnosti v takom poradí, ako sú uložené v pamäti. To samozrejme nie je vždy možné, a preto ďalším vylepšením je možnosť spracovávať inštrukcie mimo poradia – out of order execution. Tento mechanizmus sa aplikuje v prípade, že inštrukcia, ktorá ma byť vykonaná, nemá ešte pripravené argumenty v registroch procesora z dôvodu pomalšej rýchlosti pamäte. Za účelom minimalizácie času čakania, kedy sa čaká na pripravenie údajov a vykonanie inštrukcie, je možné vykonať inú inštrukciu, ktorá sa vyskytuje neskôr v inštrukčnom prúde a všetky jej argumenty sú pripravené.

Virtuálna pamäť

Použitie paralelných programov nachádza často svoje využitie práve pri spracovaní veľkého množstva údajov. Pri ukladaní dát je v operačnom systéme lineárny adresný priestor podľa potreby mapovaný do fyzickej pamäte. V takýchto prípadoch môže nastať situácia, že nie všetky údaje, s ktorými program pracuje, sa zmestia do hlavnej pamäte počítača. Okrem toho je potrebné si uvedomiť, že súčasne používaný hardvér a operačné systémy umožňujú súbežné vykonávanie viacerých programov – multitasking. Tento spočíva v rýchlom prepínaní medzi jednotlivými programami, ktorým je striedavo prideľovaný výpočtový výkon, čo vytvára dojem, že všetky programy sú vykonávané súbežne. Toto ešte viac prispieva k problémom s prácou s pamäťou, pretože ju medzi sebou musia zdieľať viaceré programy tak, aby nedošlo k poškodeniu alebo strate údajov a zároveň mal každý vykonávaný program dostatok miesta v pamäti.

Riešením je použitie virtuálnej pamäte, pomocou ktorej je možné použiť sekundárne úložisko (HDD, SSD, atď.) na rozšírenie kapacity hlavnej pamäte. Vzhľadom k tomu, že nie všetky programy bežia v tom istom čase a nie k všetkým údajom v hlavnej pamäti je pristupované v rovnakom čase, je možné niektoré bloky pamäte, označované ako stránky s rovnakou veľkosťou 4 až 16 kB, uložiť na špeciálne pripravené miesto na sekundárnom úložisku – swap priestor. Tu však narážame na problém rýchlosti prenosu údajov. Prístup k údajom uloženým na pevnom disku je o niekoľko rádov pomalší, než prístup k údajom v pamäti RAM.

Keby sme chceli stránkam priraďovať fyzickú adresu už počas prekladu programu, v multitasking podporujúcom operačnom systéme by s najväčšou pravdepodobnosťou nastal konflikt, keď by viacero programov chcelo uložiť svoje údaje na tú istú fyzickú adresu. Je zrejmé, že daná stránka na určitej fyzickej adrese môže byť používaná len jedným programom a ostatné programy by museli počkať, kým sa stránka neuvoľní. Z tohto dôvodu sa pri preklade programu priraďujú virtuálne čísla stránok. Následne sa po spustení programu vytvorí stránka, ktorá mapuje virtuálne číslo na fyzickú adresu v pamäti. Počas behu sa potom program odvoláva na virtuálnu adresu, ktorá je pomocou tabuľky stránok prevedená na konkrétnu fyzickú adresu. Keďže manažovanie tabuľky stránok je vykonávané operačným systémom, tento mechanizmus umožňuje zabezpečiť, aby sa miesta v pamäti pridelené rôznym programom neprekrývali.

Efektivita operačného systému používajúceho virtuálnu pamäť závisí aj od stupňa lokality odkazov v programoch. Lokalitu môžme posudzovať z pohľadu času a priestoru. Pokiaľ sa použije niektorá inštrukcia alebo údaj uložený v pamäti, často nastane situácia, že bude skoro použitý opätovne. Túto vlastnosť vieme využiť tak, že vhodne usporiadame programové konštrukcie, ako sú napríklad cykly, často používané premenné a podprogramy. Taktiež pokiaľ sa počas vykonávania programu použije nejaké miesto v pamäti, je pravdepodobné, že bude skoro použité aj susedné miesto v pamäti. Túto vlastnosť vieme

využiť tak, že vhodne usporiadame inštrukcie, lineárne údajové štruktúry (polia) a vyhradíme susedné miesta v pamäti pre často používané premenné. Vo všeobecnosti je potrebné snažiť sa o dosiahnutie čo najvyššieho stupňa lokality [8].

Princíp použitia virtuálnej pamäte spočíva v možnosti dočasného presunutia a uvoľnenia aktuálne nepoužívaných stránok s údajmi z hlavnej pamäti do swap priestoru na sekundárnom úložisku, pričom zmeny sú realizované prostredníctvom bufra. Cenou za výhody, ktoré nám stránkovanie ponúka je zdvojnásobenie času prístupu k požadovanému pamäťovému miestu, pretože pred samotným pristúpením k údaju na fyzickej adrese v pamäti je potrebné túto adresu získať preložením z virtuálnej adresy. Ďalším významným dôsledkom je spomalenie vykonávania samotného programu, ako aj celého operačného systému v dôsledku swapovania. Hoci programátor priamo nemôže ovplyvniť správu virtuálnej pamäte, je potrebné poznať tieto skutočnosti, aby bolo možné predchádzať zbytočnému spomaľovaniu vykonávania programu v dôsledku nadmerného využívania swap priestoru. V niektorých prípadoch v systémoch pre vysokovýkonné počítanie nie je možné dokonca swap priestor vôbec používať.

Inštrukčný paralelizmus

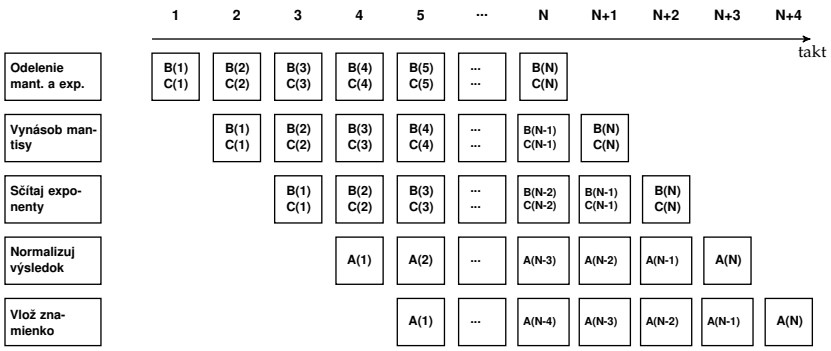
Inštrukčný paralelizmus predstavuje ďalší nástroj na dosiahnutie rýchlejšieho vykonávania programov. Jeho princíp spočíva vo využití viacerých funkčných jednotiek procesora, ktoré dokážu simultánne vykonávať inštrukcie. Dva najrozšírenejšie prístupy sú prúdové spracovanie inštrukcií – pipelining a superskalárna architektúra – superskalarita.

Pipelining predstavuje jedno z najvýznamnejších vylepšení procesora. Princíp spočíva v rozdelení zložitejších inštrukcií, ako sú napríklad operácie sčítania alebo násobenia s operandmi s pohyblivou desatinnou čiarkou, ktoré nie je možné vykonať počas jedného taktu. Takéto inštrukcie je možné rozdeliť na jednoduché časti – mikrooperácie, ktoré budú vykonané rôznymi funkčnými jednotkami procesora, čo umožní zväčšiť počet inštrukcií vykonaných za jeden takt. V ideálnom prípade je v každom takte dokončená jedna inštrukcia. Na obrázku 3 je znázornený príklad výpočtu súčinu dvoch n-prvkových vektorov s využitím pipeliningu. Inštrukcia násobenia dvoch reálnych čísel je rozdelená na päť častí, pričom každá z nich je vykonávaná inou funkčnou jednotkou, ktoré sú navzájom zreťazené.

Poznámka: Procesor Intel Pentium 4 umožňoval pipelining pozostávajúci z 20 krokov. Súčasné procesory umožňujú kratší pipelining.

Výhodou rozdelenia inštrukcií na jednoduchšie časti umožňuje aj zjednodušenie funkčných jednotiek a teda aj zrýchlenie ich taktovacej frekvencie. V praxi však zvyčajne rozdelenie inštrukcie na k častí neznamená automaticky k-násobné zrýchlenie. Dôvodom je rozdielny čas potrebný na vykonanie jednotlivých krokov, a preto je vždy rýchlosť určujúca práve najpomalšia časť, pričom ostatné funkčné jednotky zatiaľ musia čakať.

Ďalším príkladom inštrukčného paralelizmu je superskalárna architektúra. Vďaka nej je možné v tom istom takte dynamicky rozdeľovať a súbežne vykonávať viacero inštrukcií. K tomu je potrebné, aby



Obr. 3: Časový priebeh zjednodušenej operácie násobenia reálnych čísel s použitím pipeliningu

procesor obsahoval viacero rovnakých funkčných jednotiek, ktoré sú schopné pracovať súbežne. Hlavným rozdielom od viacjadrových procesorov je zvýšenie počtu iba niektorých častí procesora a nie celých jadier procesora. V prípade, že máme k dispozícii dve kompletné jednotky pre sčítanie v pohyblivej desatinnej čiarke, vedeli by sme skrátiť čas sčítania dvoch vektorov na polovicu.

K tomu, aby sme mohli profitovať z využitia superskalarity je potrebné, aby systém našiel také inštrukcie, ktoré je možné vykonať súbežne. Prekladač alebo procesor sa snaží urobiť odhad a na základe neho inštrukcie vykoná. Tento princíp si objasníme na nasledujúcom príklade zdrojového kódu 2.

Zdrojový kód 2: Ukážka odhadu špekulácie

```
1  int x, y, z;
2
3  z = x + y;
4  if(z > 0) {
5    w = x;
6  } else {
7    w = y;
8  }
```

V uvedenom príklade systém predpokladá, že výsledná hodnota v premennej `z` bude kladné číslo, a preto do premennej `w` priradí hodnotu z premennej `x`. V opačnom prípade, ak by odhad nebol správny a výsledok v premennej `z` by nebol väčší ako `0`, by bolo opätovne potrebné vykonať operáciu priradenia obsahu premennej `y` do premennej `w`.

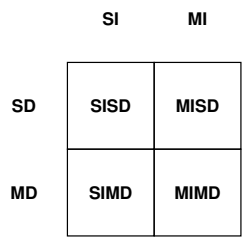
Doteraz sme sa zaoberali niektorými vylepšeniami procesorov, ktoré vedú k efektívnejšiemu vykonávaniu inštrukcií a spracovávaniu údajov. Pri niektorých z nich je zrejmé, že ide o akési paralelné spracovanie údajov, avšak takéto vykonávanie inštrukcií je pred samotným programátorom viac-menej skryté a on nemá významnú možnosť ovplyvniť spôsob vykonávania svojho programu.

2 Paralelné architektúry

V predchádzajúcej kapitole sme sa zmienili o niektorých možnostiach, ktoré nám ponúkajú architektúry dnešných procesorov, pričom ich ale programátor nemôže priamo ovplyvňovať. Naopak výber vhodnej hardvérovej architektúry pre riešenie určitého paralelného algoritmu má zásadný vplyv na efektivitu využitia hardvérových prostriedkov a v konečnom dôsledku aj na rýchlosť, s akou sa vieme dopracovať k požadovanému výsledku. Na voľbe architektúry následne záležia aj rôzne detaily samotnej implementácie a prispôsobenie návrhu softvéru.

Flynnova taxonómia

Pre klasifikáciu paralelných hardvérových architektúr je pomerné rozšírená Flynnova taxonómia [9, 10]. Základom klasifikácie je rôzny pohľad na spracovanie inštrukčného a dátového prúdu. Inštrukčný prúd predstavuje postupnosť inštrukcií vykonávaných procesorom a dátový prúd predstavuje postupnosť údajov spracovaných vykonávanými inštrukciami. Procesor môže zároveň spracovávať jeden alebo niekoľko takýchto prúdov, čo umožňuje štyri možné kombinácie znázornené na obrázku 4, pričom je potrebné podotknúť, že nie všetky sú paralelné. Spracovanie jedného prúdu inštrukcií označíme ako **SI** – Single Instruction, viacerých prúdov inštrukcií označíme **MI** – Multiple Instruction, jedného prúdu údajov označíme **SD** – Single Data a viacerých prúdov údajov označíme ako **MD** – Multiple Data.



Obr. 4: Flynnova taxonómia

Z uvedených štyroch možností spracovania inštrukčného a dátového prúdu nás budú obzvlášť zaujímať dva prípady a to **SIMD** a **MIMD**. Tieto dve architektúry patria v súčasnosti medzi najrozšírenejšie a najviac používané paralelné architektúry. Nakoľko architektúra **SISD** je schopná súčasne spracovávať len jeden prúd inštrukcií a údajov, jedná sa o sekvenčné a nie paralelné spracovanie jednojadrovým procesorom. Taktiež architektúra **MISD** umožňuje súčasné vykonávanie viacerých prúdov inštrukcií s rovnakým prúdom údajov. Táto architektúra je zvlášť vhodná na riešenie len úzkej skupiny úloh, napríklad redundancia výpočtov pre nulovú toleranciu chýb [11], a preto nie je v praxi veľmi rozšírená a bežne komerčne dostupná.

Architektúra SIMD

Táto paralelná architektúra umožňuje súčasné vykonávanie tej istej inštrukcie na rôznych množinách údajov, vďaka viacerým aritmeticko-logickým jednotkám (ALJ) ovládaných jednou riadiacou jednotkou, pričom každá aritmeticko-logická jednotka má svoju lokálnu pamäť obsahujúcu nejaký údaj. Táto architektúra je vhodná na riešenie paralelných problémov, kde je potrebné vykonávať rovnakú operáciu na veľkej množine údajov, ako napríklad spracovanie obrazu, operácie s vektormi atď. Problém môže nastať v prípade, ak sú spracovávané

údaje rozdielne do takej miery, že ich spracovanie jednotlivými procesormi trvá rôzne dlho, a preto je potrebné procesory po každom kroku synchronizovať a počkať na najpomalší z nich [12].

Ako príklad si uveďme sčítanie dvoch vektorov x a y s dĺžkou n prvkov, pričom výsledok chceme uložiť do vektora x, viď. zdrojový kód 3.

Zdrojový kód 3: Kód pre sčítanie dvoch vektorov

```
1  for(i = 0; i < n; i++) {
2    x[i] += y[i];
3  }
```

Za predpokladu, že by náš systém disponoval viacerými ALJ ako je počet prvkov vektora n, tak by bolo možné operáciu sčítania týchto vektorov vykonať v jednom kroku tak, že i-tá ALJ by vykonala operáciu sčítania prvkov `x[i]` a `y[i]`. V prípade, ak by vektor obsahoval viac prvkov ako je počet ALJ, tak by bolo potrebné postup niekoľkokrát zopakovať.

Problematická situácia nastáva v prípade vetvenia programu, pretože všetky ALJ musia vykonávať tú istú inštrukciu. V takomto prípade je možné využiť len niektoré z výsledkov, zatiaľ čo výsledok ALJ vykonávajúcich opačnú vetvu sa nepoužijú. Následne sú vykonané inštrukcie z opačnej vetvy. V dôsledku tohto je využitá iba časť aritmeticko-logických jednotiek, čo vedie k výraznému zníženiu výpočtového výkonu procesora. Ako môžeme vidieť, takáto forma dátového paralelizmu je vhodná najmä pre paralelizáciu cyklov vykonávajúcich operácie s vektormi, kde sú s jednotlivými údajmi vykonávané viac menej tie isté inštrukcie.

Systémy s architektúrou SIMD sú zvlášť efektívne pri spracovaní paralelných problémov s veľkým množstvom údajov. Začiatkom 90-tych rokov 20. storočia bol najväčším výrobcom výrobca SIMD systému (Thinking Machine) [13]. Avšak ku koncu 20. storočia boli jedinými vyrábanými systémami so SIMD architektúrou vektorové procesory. V nedávnej minulosti boli prvky tejto architektúry použité aj v grafických kartách (GPU) a desktopových CPU [4].

Zatiaľ čo bežný procesor vykonáva operácie s jednotlivými prvkami údajov, **vektorový procesor** vykonáva operácie s vektormi alebo poliami údajov. Prekladač pre takéto procesory dokáže veľmi efektívne vyhľadávať časti kódu, ktorý je možné vektorizovať. Tiež dokáže identifikovať cykly, pri ktorých vektorizáciu nie je možné použiť a na základe zdôvodnenia je možné zvážiť upravenie zdrojového kódu tak, aby bolo možné vektorizáciu použiť. Takéto systémy disponujú veľkou prenosovou šírkou pre komunikáciu s pamäťou. Všetky načítané údaje z pamäte sa použijú pri výpočtoch, a preto nie je možné pracovať s nepravidelnými údajovými štruktúrami.

Špeciálnym prípadom hardvéru, ktorý je postavený na princípe architektúry SIMD, sú **grafické karty**. Ich primárnou úlohou je pretransformovať vnútornú reprezentáciu objektov pomocou bodov, čiar a

trojuholníkov na množinu bodov odpovedajúcim zobrazovaným pixelom. Procesor GPU pozostáva z veľkého množstva jednoduchých jadier, ktoré zväčša vykonávajú tie isté operácie, čo umožňuje ich efektívne využitie. Pri spracovaní obrazu je potrebné pracovať s veľkým množstvom údajov, a preto GPU disponuje komplexnou viacúrovňovou hierarchiou pamäte. Súčasne GPGPU (General Purpose GPU) je možné využiť aj pre iné paralelné výpočty ako spracovanie obrazu. Tieto obsahujú niekoľko desiatok multiprocesorov, pričom každý z nich je schopný vykonávať odlišné operácie. V súčasnosti predstavujú GPGPU významnú časť dostupných prostriedkov pre vysokovýkonné počítanie poskytované superpočítačovými centrami.

Architektúra MIMD

Architektúra MIMD umožňuje súčasné vykonávanie viacerých prúdov inštrukcií na viacerých prúdoch údajov, a teda umožňuje vykonávať súčasne aj rôzne programy. V porovnaní so SIMD architektúrou, architektúra MIMD pozostáva z menšieho počtu procesorov alebo jadier, avšak tieto sú navzájom úplne nezávislé. To znamená, že každý z nich má svoju vlastnú riadiacu aj aritmeticko-logickú jednotku. Okrem toho všetky procesory pracujú asynchrónne s nezávislým časovaním. Podrobnejšie sa budeme venovať týmto systémom v podkapitole 3.

3 Viacjadrové procesory

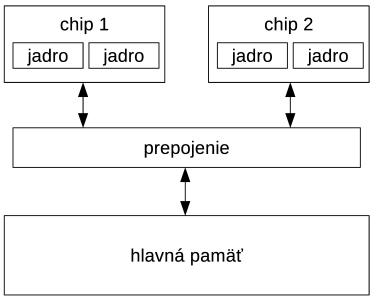
Ako sme sa už zmienili v kapitole **??**, v istom okamihu nastal zlom v možnosti zvyšovania rýchlosti jednojadrových procesorov, a to ako v zvyšovaní ich taktovacej frekvencie, tak aj v zvyšovaní ich integrácie. Dôvodom bola neúmerná spotreba energie a následná potreba chladenia procesora, ale aj fakt, že prepojenia medzi jednotlivými tranzistormi integrovaného obvodu sa blížili k veľkosti len niekoľkých atómov. Riešením bol nástup **viacjadrových procesorov**. Tieto pozostávajú z viacerých jadier umiestnených v rámci jedného procesora, pričom sú rozložené na väčšej ploche v porovnaní s konvenčným procesorom. Každé jadro procesora dokáže pracovať so svojimi údajmi a vykonávať svoje inštrukcie. Architektúra takéhoto procesora sa môže odlišovať v závislosti blízkosti prepojenia jadier, použití vyrovnávacej pamäte, funkčných jednotiek a ďalších komponentov, a či môžu byť tieto komponenty využívané samostatným jadrom alebo sú zdieľané medzi určitou skupinou alebo všetkými jadrami.

Vzhľadom na možnosť prístupu do pamäte môžme hovoriť o dvoch typoch systémov:

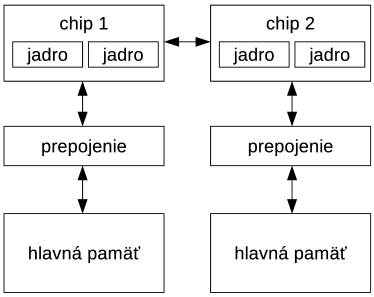
- systém so **zdieľanou pamäťou** (shared memory computer),
- systém s **distribuovanou pamäťou** (distributed memory computer).

Systém so zdieľanou pamäťou

Systém so zdieľanou pamäťou umožňuje vykonávať operácie viacerým nezávislým procesorom nad spoločným, zdieľaným fyzickým adresným priestorom. V dôsledku toho môže každý procesor pristupovať k údajom uloženým kdekoľvek v celej pamäti. Najčastejšie riešenie tohto typu predstavuje viacjadrový procesor. Hoci sa prístup k takejto pamäti pre jednotlivé jadrá procesora môže programátorovi javiť ako jednotný, nemusí tomu tak naozaj byť. Existujú dva typy systémov so zdieľanou pamäťou, ktoré majú významný vplyv na rýchlosť prístupu k údajom uloženým v hlavnej pamäti. Ide o systémy s viacerými viacjadrovými procesormi, v ktorých môžu byť všetky procesory prepojené s hlavnou pamäťou, alebo každý z procesorov disponuje prepojením s určitou časťou hlavnej pamäte a prístup k ostatným častiam zabezpečuje špecializovaný hardvér, ktorý je súčasťou procesora. Tento rozdiel nie je pre programátora viditeľný z pohľadu možnosti prístupu k údajom uloženým v pamäti, avšak je badateľný z pohľadu času prístupu k údajom.



Obr. 5: Schéma viacjadrového systému so zdieľanou pamäťou s UMA



Obr. 6: Schéma viacjadrového systému so zdieľanou pamäťou s NUMA

V prípade, že čas prístupu a prenosová rýchlosť k akémukoľvek bloku pamäte z ľubovoľného jadra procesora je rovnaký, jedná sa o architektúru s jednotným prístupom k pamäti **(Uniform Memory Access – UMA)**, znázornená na obrázku 5. Multiprocesor, v ktorom sú všetky procesory homogénne s rovnako rýchlym prístupom k ďalším prostriedkom, ako sú vstupno-výstupné zariadenia, zbernice, atď., sa nazývajú **symetrické multiprocesory** (Symmetric Multiprocessors – SMP).

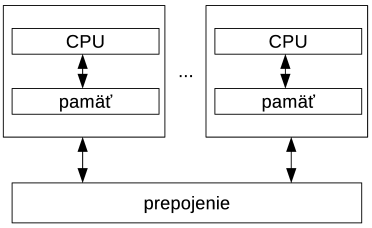
V druhom prípade je čas prístupu k blokom pamäte, ktoré sú priamo prepojené s jadrom, kratší a prenosové rýchlosť vyššia v porovnaní s prístupom k ostatným blokom pamäte. V takomto prípade ide o architektúru s nejednotným prístupom k pamäti **(Nouniform Memory Access – NUMA)**, znázornená na obrázku 6. Hoci sa táto architektúra na prvý pohľad skôr podobá na systém s distribuovanou pamäťou, vnútorné prepojenie zabezpečuje, že celá agregovaná pamäť sa javí ako jeden adresný priestor. Zásadný rozdiel je však v rozličnej rýchlosti prístupu k jednotlivým častiam pamäte.

Komunikáciu medzi jednotlivými jadrami je možné realizovať pomocou zdieľanej pamäte, pričom jedno jadro môže údaje do pamäte zapísať, zatiaľ čo druhé jadro ich môže následne prečítať. Spravidla je jednoduchšie navrhnúť a implementovať program pre architektúru UMA, pretože programátor sa nemusí zaoberať umiestnením dát a rýchlosťou k ich prístupu. Naopak architektúra NUMA zas poskytuje rýchlejší prístup k údajom uloženým v priamo pripojených blokoch pamäte, ako aj možnosť práce s potenciálne vyššou kapacitou pamäte RAM v porovnaní s architektúrou UMA.

Systém s distribuovanou pamäťou

V systémoch s distribuovanou pamäťou disponuje každý procesor svojou vlastnou lokálnou pamäťou, do ktorej ostatné procesory nemajú prístup. Jednotlivé procesory môžu navzájom komunikovať pomocou

prepojovacej siete znázornenej na obrázku 7. Najbežnejším príkladom systému s distribuovanou pamäťou je počítačový **klaster**. Tento zväčša pozostáva, z dôvodu cenovej výhodnosti, z väčšieho množstva komoditných systémov navzájom prepojených komunikačnou sieťou. Jednotlivé uzly klastra sú spravidla samostatné, obyčajné počítače so zdieľanou pamäťou, pričom každý z nich obsahuje minimálne jedno sieťové rozhranie, pomocou ktorého sú uzly navzájom prepojené. Takéto systémy niekedy označujeme aj ako **hybridné systémy**.



Obr. 7: Schéma systému s distribuovanou pamäťou

Keďže táto architektúra neumožňuje prístup všetkých procesorov k rovnakej pamäti tak, aby si pomocou nej mohli jednotlivé procesory vymieňať informácie, je potrebné túto komunikáciu zabezpečiť pomocou komunikačnej siete. Táto komunikácia je spravidla implementovaná pomocou posielania správ medzi jednotlivými procesmi, ktoré spolupracujú na riešení rovnakej úlohy. Najviac používaný štandard pre komunikáciu pomocou posielania správ je **MPI** (Message Passing Interface), ktorý poskytuje širokú paletu možností v porovnaní s paradigmou programovania pre systémy so zdieľanou pamäťou. Zároveň je možné použiť program implementovaný pomocou MPI aj na samotnej architektúre so zdieľanou pamäťou. Samotná implementácia programov pre systémy s distribuovanou pamäťou je náročnejšia z dôvodu, že programátor musí mať dobrý prehľad o umiestnení používaných údajov na jednotlivých uzloch, pričom akákoľvek výmena údajov medzi procesmi alebo ich synchronizácia musí byť realizovaná pomocou poslanej správy medzi dvoma alebo viacerými zúčastnenými procesmi.

Väčšinu súčasných vysokovýkonných systémov vo svete predstavujú práve počítačové klastre a superpočítače. V prípade, že sú jednotlivé distribuované počítače geograficky vzdialené a predstavujú heterogénny systém, tak ich označujeme ako počítačový **grid**. V rámci gridu sa vyskytuje veľké množstvo rozličných počítačov a rôznych architektúr.

Niektoré systémy zas môžu pozostávať z väčšieho množstva procesorov agregovaných v rámci jedného systému. Zväčša ide o špecializovaný proprietárny hustejšie integrovaný hardvér, určený na riešenie výpočtovo veľmi náročných úloh. Vyššia hustota, množstvo procesorov a rýchlejšie prepojenia medzi výpočtovými uzlami zabezpečujú lepšiu a rýchlejšiu komunikáciu medzi procesmi a vyšší agregovaný výkon. Takéto systémy označujeme ako **systémy pre masívne paralelné spracovanie** (Massively Parallel Processing – MPP) alebo **superpočítače** [4, 14].

Poznámka: Označenie systémov ako MPP nie je v literatúre jednoznačne definované. Za MPP systémy je možné považovať aj vysokovýkonné klastre vybudované z proprietárnych komponentov, ktoré umožňujú škálovanie do veľkého počtu uzlov.

4 Počítačový klaster

Vysokovýkonné počítačové klastre predstavujú najbežnejšie dostupné prostriedky pre riešenie výpočtovo náročných úloh. Ich hlavnou výhodou je relatívna jednoduchosť a dobrý pomer výkonu voči potrebným nákladom. Z pohľadu programátora takýto systém ponúka paralelizácia úloh na dvoch úrovniach [15]:

- viacjadrový procesor v rámci jedného uzla, prípadne ďalší akcelerátor (GPGPU, Intel Xeom Phi),
- veľké množstvo uzlov spravidla prepojených pomocou siete Gbit Ethernet alebo Infiniband s veľmi nízkou latenciou a vysokou prenosovou rýchlosťou.

Poznámka: Sieť typu Infiniband sa vyznačuje nízkou latenciou približne 1-5 μs a vysokou prenosovou rýchlosťou okolo 300 Gbit/s pre EDR s 12 linkami.o

Jednotlivé výpočtové uzly klastra predstavujú samostatné autonómne počítačové podsystémy, ktoré navzájom komunikujú pomocou posielania správ cez komunikačnú prepojovaciu sieť. Zároveň jednotlivé procesory v rámci jedného uzla môžu pre vzájomnú komunikáciu využiť zdieľanú pamäť v spoločnom adresnom priestore. V dôsledku toho je komunikácia pomocou zbernice medzi jednotlivými procesormi v jednom uzle asi 10 až 1000-krát rýchlejšia v porovnaní s komunikáciou medzi jednotlivými uzlami.

V súčasnosti je podľa rebríčka TOP500 [16] z novembra 2019 najvýkonnejších systémov na celom svete najvýkonnejší systém SUMMIT v IBM DOE/SC/Oak Ridge National Laboratory v USA, založený na technológiách firiem IBM, NVIDIA a Mellanox. Je tvorený takmer 2,5 miliónmi jadier a dosahuje maximálny výkon viac než 148 PFlop/s pri spotrebe elektrickej energie približne 10 MW. Náklady na obstaranie a prevádzku takýchto superpočítačov sú však enormné, a preto si ich môže dovoliť prevádzkovať len niekoľko dobre financovaných výskumných laboratórií a vládnych organizácií.

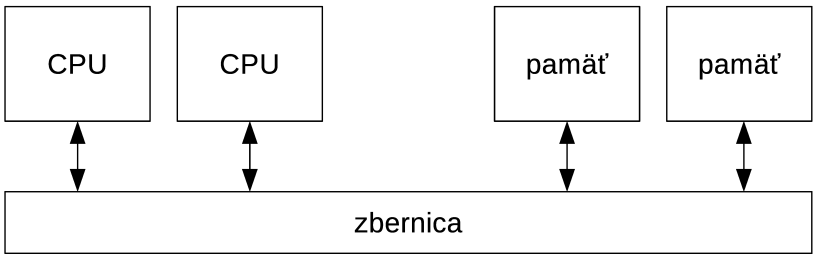
Alternatívu k nákladným riešeniam v podobe superpočítačov predstavuje klaster zložený z bežne dostupného hardvéru, napríklad používaného v osobných stolových počítačoch. Prvý takýto klaster sa objavil v roku 1994 v spoločnosti NASA a bol pomenovaný Beowulf. V súčasnosti **Beowulf klaster** predstavuje synonymum pre menej nákladné riešenie počítačového klastra postaveného zo štandardných komponentov bežne dostupných na trhu. Výhodou tohto riešenia je vysoký výkon za relatívne nízke náklady, flexibilita a možnosť prispôsobenia potrebám, jednoduché rozšírenie a modernizácia komponentov klastra a široká škála použiteľných hardvérových a softvérových komponentov. Množstvo používaných nástrojov, ako aj samotný operačný systém predstavujú voľne dostupné riešenia s otvorenými zdrojovými kódmi vyvíjaných v rámci Projektu GNU [17]. Vysoká miera škálovateľnosti od niekoľkých po niekoľko desiatok tisícov jadier umožňuje jeho široké uplatnenie v praxi [14].

5 Prepojovacie siete

Paralelné programovanie umožňuje rozdelenie výpočtovo náročných úloh na podúlohy, ktoré je možné vykonávať súbežne viacerými procesmi. Tieto však musia spolu komunikovať, čo zohráva významnú rolu v efektívnosti vykonávania paralelných programov, ako pri použití zdieľanej, tak aj distribuovanej pamäte. Akákoľvek ich vzájomná komunikácia vedie spravidla k znižovaniu výkonnosti paralelného programu. Preto je potrebné si objasniť základné možnosti ich vzájomného prepojenia.

Systémy so zdieľanou pamäťou

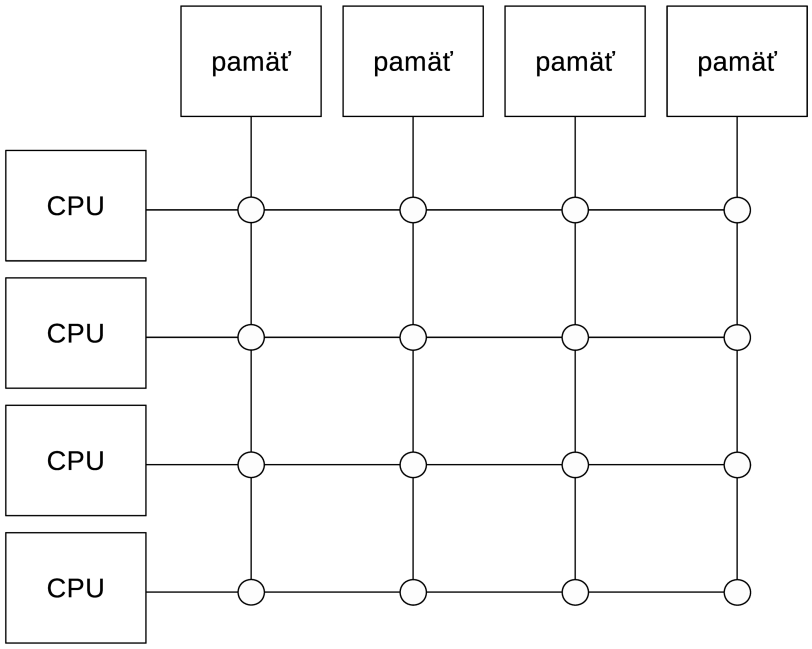
V systémoch so zdieľanou pamäťou je potrebné zabezpečiť komunikáciu jednotlivých procesorov s modulmi pamäte RAM. Na realizáciu tohto prepojenia sa zväčša používa buď komunikačná zbernica alebo prepínač s krížovou architektúrou. Princíp zbernice spočíva v použití spoločných paralelných prepojení, ku ktorým sú pripojené všetky zariadenia tak, ako je znázornené na obrázku 8.



Obr. 8: Zbernicové prepojenie CPU a RAM

Zbernica umožňuje pomerne jednoduché pripojenie ďalšieho zariadenia a tým umožňuje veľkú variabilitu. Nevýhodou potom je fakt, že pri pripojení väčšieho množstva zariadení ku spoločnej zbernici všetky tieto zariadenia musia zdieľať spoločný komunikačný kanál, a tak dochádza častejšie k jej vyťaženiu, v dôsledku čoho musia zariadenia častejšie čakať a celkový výkon sa znižuje.

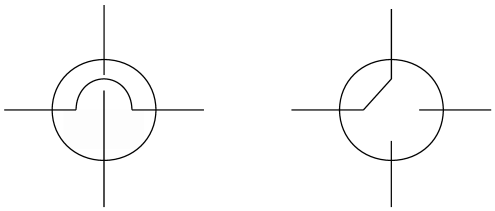
V prípade použitia väčšieho systému so zdieľanou pamäťou sa preto častejšie vyskytuje prepojenie pomocou prepínača s krížovou architektúrou znázorneného na obrázku 9, ktorý pozostáva z procesorov, pamäťových modulov, prepojení a prepínačov.



Obr. 9: Prepojenie CPU a RAM pomocou prepínača s krížovou architektúrou

Takáto architektúra obsahuje viacero prepínačov, ktoré riadia prístup jednotlivých procesorov k pamäťovým modulom. Každý z prepína-

čov sa môže nachádzať v dvoch rôznych stavoch znázornených na obrázku 10.



Obr. 10: Dva možné stavy prepínača

V prípade použitia rovnakého alebo väčšieho počtu pamäťových modulov ako je počet procesorov sa môže konflikt vyskytnúť len v prípade, že sa dva procesory budú snažiť súčasne pristupovať k rovnakému pamäťovému modulu. Keďže takéto prepojenie umožňuje súbežnú komunikáciu viacerých zariadení, tým pádom poskytuje vyššiu rýchlosť komunikácie v porovnaní so zbernicou.

Systémy s distribuovanou pamäťou

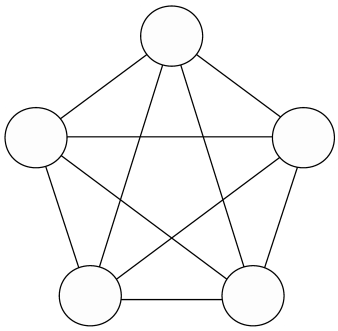
V systémoch s distribuovanou pamäťou si jednotlivé procesory vymieňajú medzi sebou údaje pomocou prepojovacej siete. Takáto komunikácia môže prebiehať buď synchrónne alebo asynchrónne v podobe posielania správ medzi procesormi cez komunikačnú sieť. V prepojovacích sieťach môže byť každý procesor s pamäťou priamo prepojený s prepínačom, vtedy hovoríme o **priamom prepojení**. Spôsob, akým sú jednotlivé procesory navzájom prepojené, môže byť odlišný a predstavuje **topológiu prepojovacej siete**, ktorá môže byť znázornená pomocou grafu. Vrcholy grafu predstavujú jednotlivé procesory s lokálnou pamäťou alebo jednotlivé výpočtové uzly a hrany grafu predstavujú ich vzájomné obojsmerné komunikačné prepojenia.

Jednotlivé topológie disponujú rôznymi parametrami a sú vhodné pre rôzne typy úloh. Pre ich porovnanie je možné použiť niekoľko kritérií, ako sú priemer, stupeň, šírka bisekcie, prepojenosť hrán a náklady [14]. **Priemer** prepojovacej siete predstavuje najväčšiu vzdialenosť (počet hrán grafu) medzi dvoma ľubovoľnými procesormi v sieti. Z tohto pohľadu je najvýhodnejšia taká sieť, ktorá má čo najmenší priemer, pretože je najvýhodnejšie, keď je komunikácia medzi procesormi smerovaná cez čo najmenší počet prepojení (hrán grafu). **Stupeň** prepojovacej siete predstavuje najväčší počet prepojení jedného vrcholu grafu. Čím nižší je stupeň prepojovacej siete, tým jednoduchšie a rýchlejšie je smerovanie komunikácie, pretože je potrebné obsluhovať menší počet komunikačných kanálov. V dobrej prepojovacej sieti by priemer siete nemal s pribúdajúcim počtom procesorov p narastať viac než logaritmicky a jej stupeň by mal zostať konštantný a zároveň malý.

Ďalším kritériom pre hodnotenie kvality prepojovacej siete je **šírka bisekcie**, ktorá zodpovedá minimálnemu počtu hrán grafu, ktoré je potrebné odstrániť, aby sa sieť rozdelila na dve rovnaké podsiete. Následne súčin šírky bisekcie a prenosovej rýchlosti prepojenia označujeme ako **prenosová rýchlosť bisekcie**, ktorá zodpovedá rýchlosti prenosu údajov medzi dvoma polovicami siete. **Prepojenosť hrán**

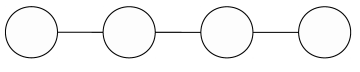
predstavuje počet existujúcich ciest medzi jednotlivými vrcholmi alebo najmenší počet hrán, ktoré musíme odstrániť, aby sa graf stal nespojitý. V neposlednom rade treba brať do úvahy aj samotnú nákladnosť prepojovacej siete, ktorá pri istom zjednodušení zodpovedá celkovému počtu potrebných prepojení použitých v prepojovacej sieti.

Za ideálnu prepojovaciu sieť z hľadiska rýchlosti je možné považovať sieť predstavujúcu úplný graf znázornený na obrázku 11. V takejto sieti sa nachádza priame prepojenie medzi každou dvojicou procesorov, čo umožňuje ich rýchlu a efektívnu komunikáciu. Avšak v praxi použitie takýchto sietí nie je vhodné, nakoľko s rastúcim počtom procesorov počet potrebných prepojení narastá kvadraticky a stupeň vrcholu grafu narastá lineárne, v čoho dôsledku sú náklady na sieť príliš vysoké. Preto sa zväčša pristupuje k určitému kompromisu medzi počtom prepojení medzi procesormi a nákladmi na prepojovaciu sieť.

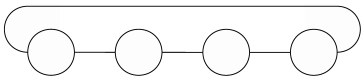


Obr. 11: Úplná prepojovacia sieť

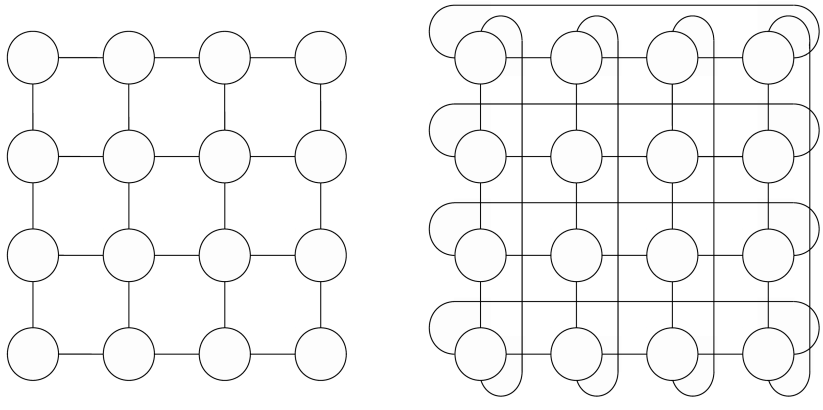
Určitý kompromis predstavujú prepojovacie siete v tvare mriežky, kde v k-rozmernej mriežke s vrcholmi zoradenými v rade je každý vrchol prepojený s ďalšími $2k$ susednými vrcholmi s výnimkou dvoch koncových vrcholov. Takáto sieť typu **mesh** je znázornená na obrázku 12. Vzájomným prepojením dvoch krajných vrcholov získame prepojovaciu sieť typu **ring** (torus) znázornenú na obrázku 13. Výhodou takejto 1-rozmernej siete je malý stupeň vrcholu o hodnote 2 a jednoduchá škálovateľnosť siete. Naopak nevýhodou pri použití veľkých sietí je ich veľký priemer. Na obrázku 14 sú znázornené 2-rozmerné prepojovacie siete typu mesh a torus. Takáto 2-rozmerná prepojovacia sieť, kde $k = 2$ so stupňom vrchola 4 má priemer $O(\sqrt{p})$.



Obr. 12: Prepojovacia sieť typu 1-rozmerný mesh



Obr. 13: Prepojovacia sieť typu 1-rozmerný ring



Obr. 14: Prepojovacia sieť typu 2-rozmerná (a) mesh a (b) torus

Prepojovacie siete typu 2- a 3-rozmerných mriežok majú dobré uplatnenie pri úlohách, kde sa pracuje s pravidelnými štruktúrami, pretože sa na takúto sieť dobre mapujú. Ako príklad možno spomenúť výpočty v lineárnej algebre s maticami alebo v počítačovej grafike, kde je možné obraz pravidelne rozdeliť na spracovanie jednotlivými uzlami siete [14].

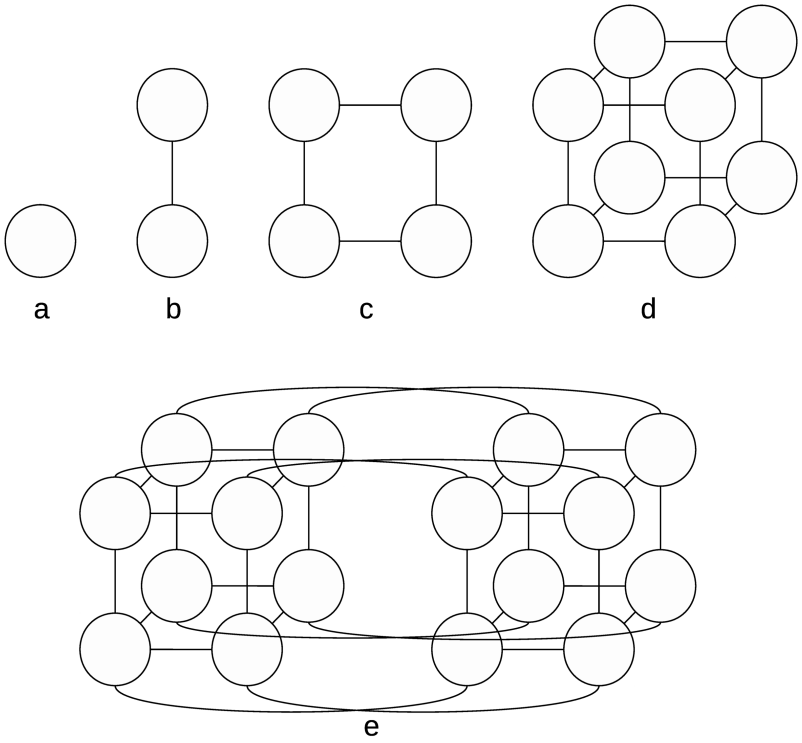
Iným prípadom bežne používaného kompromisu je vysoko prepojená sieť typu **hyperkocka**. Takáto k-rozmerná hyperkocka obsahuje počet procesorov $p = 2^k$ a každý vrchol je stupňa $\log p$. Priemer takejto

Úloha: Určte šírku bisekcie prepojovacej siete typu planárnej štvorcovej mriežky.

Úloha: Určte šírku bisekcie prepojovacej siete typu 3-rozmernej mriežky.

Úloha: S použitím induktívnej definície hyperkocky zdôvodnite, prečo je šírka bisekcie hyperkocky $p/2$.

siete je zhodný s jej rozmerom $k = \log p$. Na obrázku 15 sú znázornené prepojovacie siete typu 0- až 4-rozmerná hyperkocka. 0-rozmerná hyperkocka je kocka, ktorú predstavuje jeden vrchol. 1-rozmerná hyperkocka je tvorená dvoma 0-rozmernými hyperkockami navzájom prepojenými hranou. 2-rozmerná hyperkocka je tvorená dvoma 1-rozmernými hyperkockami navzájom prepojenými hranami na odpovedajúcich vrcholoch. Takýmto spôsobom by sme vedeli vytvoriť aj viacrozmerné prepojovacie siete typu hyperkocka.

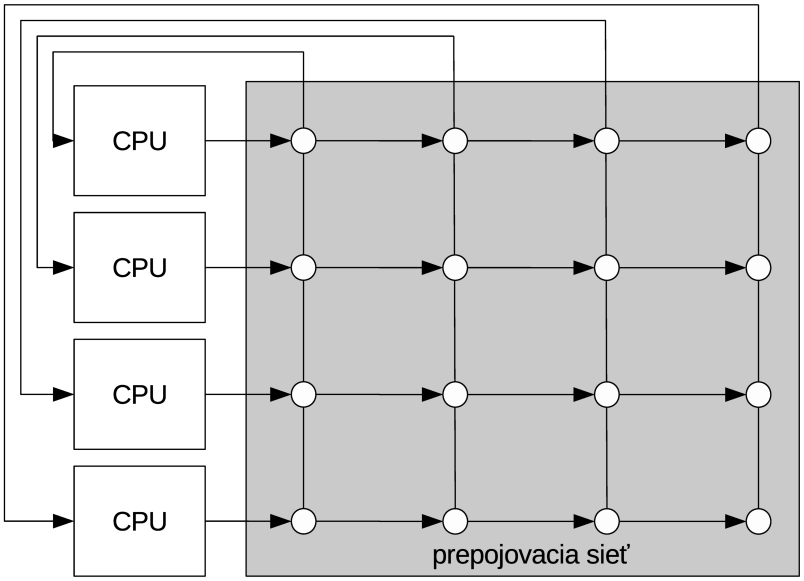


Obr. 15: Prepojovacia sieť typu n-rozmerná hyperkocka (a) n=0, (b) n=1, (c) n=2, (d) n=3, (e) n=4

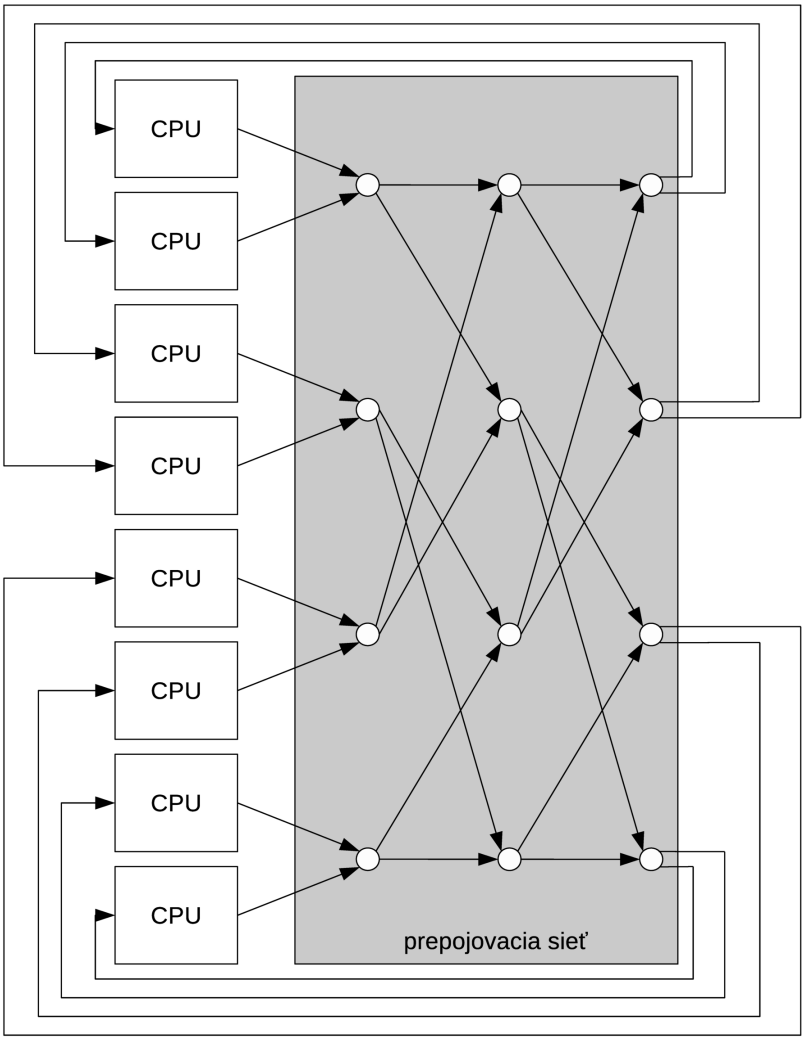
V porovnaní s prepojovacími sieťami typu mesh alebo ring sú siete typu hyperkocka efektívnejšie z hľadiska prepojenosti, ale naopak prepínače sú komplikovanejšie, nakoľko musia pracovať s viac prepojeniami v porovnaní so sieťami typu mesh alebo ring. V dôsledku toho sú tieto siete nákladnejšie.

Úloha: Šírku bisekcie prepojovacej siete s nepriamym prepojením je možné určiť tak, že procesory rozdelíme do dvoch skupín, pričom každá z nich bude obsahovať polovicu procesorov. Minimálny počet prepojení, ktoré je potrebné odstrániť aby dve polovice neboli prepojené predstavuje šírku bisekcie. Zdôvodnite, prečo je šírka bisekcie siete s krížovou architektúrou s rozmerom 8 x 8 rovná alebo menšia ako 8. Tiež zdôvodnite, prečo je šírka bisekcie prepojovacej siete typu omega s 8 procesormi rovná alebo menšia ako 4. Dve jednosmerné prepojenia považujte za jedno prepojenie.

Alternatívou k prepojovacím sieťam s priamym prepojením sú siete s **nepriamym prepojením**. V takomto prípade prepínač nie je priamo pripojený k procesoru s pamäťou. Procesor disponuje vstupným a výstupným prepojením, ktoré sú prepojené jednosmernými prepojeniami s prepojovacou sieťou. Jednoduchými ukážkami tejto topológie sú prepojovacie siete s krížovou architektúrou typu **crossbar**, znázornená na obrázku 16 alebo **omega**, znázornená na obrázku 17. Prepojovacia sieť s podobnou topológiou už bola zmienená v časti o systémoch so zdieľanou pamäťou s rozdielom, že tentokrát sú použité iba jednosmerné komunikačné prepojenia. V prípade, že žiadne dva procesory nekomunikujú s tým istým procesorom, každý procesor môže komunikovať s druhým procesorom v tom istom čase bez toho, aby sa navzájom obmedzovali. Lacnejšie riešenie, za cenu istých obmedzení, poskytuje prepojovacia sieť typu omega.



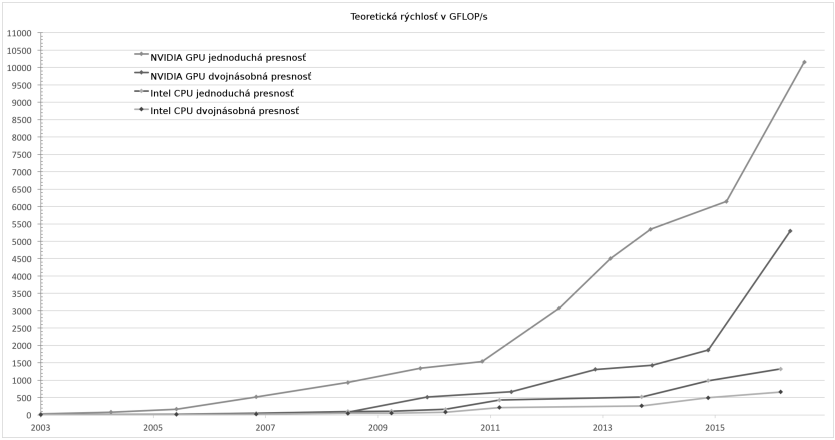
Obr. 16: Prepojovacia sieť typu crossbar



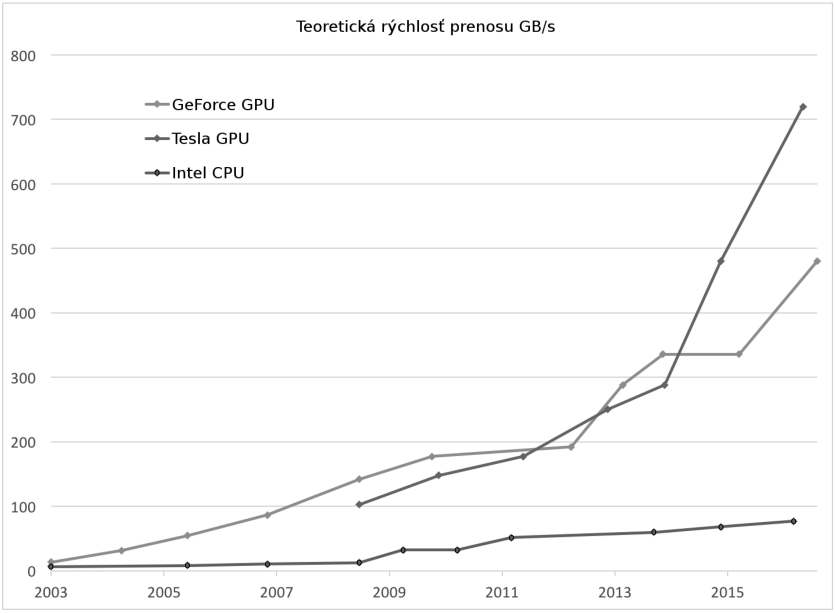
Obr. 17: Prepojovacia sieť typu omega

6 Akcelerátory GPGPU

Primárny význam grafickej karty (GPU – Graphical Processing Unit) v počítači je zabezpečiť zobrazovanie požadovaných informácií na obrazovke monitora alebo iného zariadenia. Vzhľadom na kontinuálne zvyšovanie požiadaviek na kvalitu grafického výstupu, ako plynulosť zobrazovania, vysoké rozlíšenie alebo zobrazovanie 3D scén, boli na grafické karty kladené čoraz náročnejšie výpočtové požiadavky. Pripomeňme si, že úlohou grafickej karty je vypočítať farbu a odtieň jednotlivých bodov, z ktorých sa zobrazovaný obraz skladá. Výsledkom toho je, že grafické karty sa vyvinuli do vysoko paralelných, mnohojadrových a multivláknových procesorov s vysokým výpočtovým výkonom a vysokou priepustnosťou komunikácie s pamäťou. Na obrázku 18 a 19 je znázornený vývoj rýchlosti CPU a GPU v počte vykonaných operácií s pohyblivou desatinnou čiarkou za sekundu a porovnanie vývoja priepustnosti pamäte CPU a GPU [18].



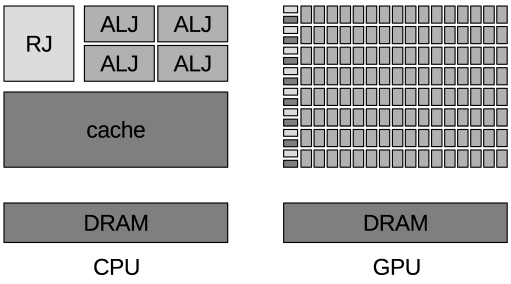
Obr. 18: Vývoj rýchlosti CPU a GPU v počte operácií s pohyblivou desatinnou čiarkou



Obr. 19: Vývoj priepustnosti pamäte CPU a GPU

Zásadný rozdiel vo výkone je spôsobený tým, že procesor GPU je navrhnutý na vykonávanie vysoko paralelných výpočtov a k tomu je prispôsobená aj jeho architektúra. Väčšia časť procesora je určená na

paralelné spracovávanie údajov, ktorá je riadená spoločnou riadiacou jednotkou a disponuje malou vyrovnávacou pamäťou. Pre porovnanie je na obrázku 20 znázornená zjednodušená štruktúra architektúry CPU a GPU.



Obr. 20: Architektúra CPU a GPU

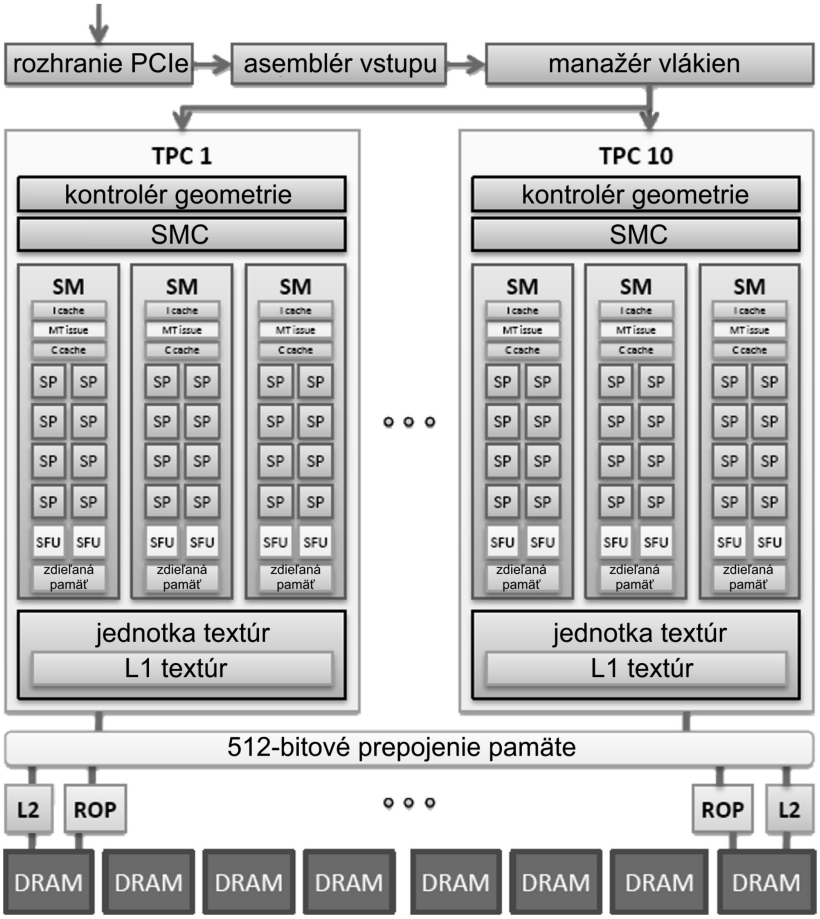
Poznámka: Aritmetickou intenzitou označujeme pomer počtu inštrukcií vykonávaných s operandmi uloženými priamo v pamäti multiprocesora k počtu inštrukcií vykonávaných s operandmi uloženými v globálnej pamäti zariadenia GPGPU. Keďže pre prístup k údajom uloženým v globálnej pamäti zariadenia sa vyznačuje vyššou latenciou, túto je pri vysokej aritmetickej intenzite možné schovať za vykonávanie ďalších inštrukcií, ktoré majú údaje pripravené v pamäti multiprocesora.

Potenciál tohto hardvéru sa ukázal byť veľmi užitočný v oblasti zobrazovania, ale aj pri vykonávaní iných výpočtovo veľmi náročných úloh. Architektúra GPU je obzvlášť vhodná pre údajový paralelizmus, kde je potrebné paralelne vykonávať ten istý program nad rôznymi časťami údajov. Keďže sa nad všetkými časťami údajov vykonávajú tie isté inštrukcie, nie je za potreby zložitého riadenia. Taktiež latencia prístupu do pamäte nie je až taká zásadná, pretože údaje sú spracovávané s vysokou aritmetickou intenzitou, v čoho dôsledku nie je potrebná ani veľká vyrovnávacia pamäť [18].

V roku 2007 predstavila spoločnosť NVIDIA platformu a programovací model **CUDA** (Compute Unified Device Architecture), ktorý umožnil relatívne jednoducho implementovať program využívajúci údajový paralelizmus na **GPGPU** (General Purpose Grpahic Processing Unit), a tak riešiť veľké množstvo výpočtovo náročných úloh oveľa efektívnejšie ako s použitím CPU. Od tejto doby vývoj GPGPU nadďalej pokračoval a vzniklo viacero architektúr GPGPU schopných vykonávania CUDA programov (Fermi, Tesla, Kepler, Maxwell, Pascal, Volta, Turing). Tieto sa navzájom môžu líšiť v počte a schopnostiach procesorov a v ďalších parametroch, avšak základný princíp architektúry je u všetkých zariadení podobný. Na obrázku 21 je znázornená architektúra Tesla.

Základným stavebným prvkom architektúry sú prúdové multiprocesory (SM – Stream Multiprocessor). Viacero SM je zoskúpených do väčších blokov. Takýchto blokov obsahuje GPGPU niekoľko. Každý SM je tvorený viacerými prúdovými procesormi (SP - Stream Processor), ktoré majú spoločnú riadiacu jednotku, inštrukčnú cache a malú zdieľanú pamäť s kapacitou niekoľko kB. Každá GPGPU je vybavená aj veľkou globálnou pamäťou DRAM s vysokou priepustnosťou a kapacitou rádovo GB, čo umožňuje benefitovať pri masívne paralelných výpočtoch [19].

Pri písaní programov pre CUDA je možné si vybrať niektorý z jazykov, ktoré sú podporované (C, C++, Fortran), okrem toho je možné použiť volania funkcií z jazykov Java alebo Python, alebo využívať technológie ako Direct compute alebo OpenACC direktívy. Programátor môže využiť aj hotové funkcie z knižnic v rôznych oblastiach, ako napríklad FFT, BLAS, RAND, SPARSE, DNN, CULA MAGMA, SVM, PhysX, OptiX, iRay alebo Matlab a Mathematica [18].



Obr. 21: Architektúra GPGPU NVIDIA Tesla C1060

Pri porovnaní viacjadrového CPU a mnohojadrového GPGPU je jasné, že obe architektúry poskytujú možnosť vykonávania paralelného programu. Zásadný rozdiel je však v škálovaní týchto paralelných programov, pričom pri viacjadrovom CPU je potrebné, aby program škáloval v rámci niekoľkých jadier CPU, zatiaľ čo pri GPGPU je potrebné, aby bol schopný transparentne škálovať v rámci rádovo stoviek až tisícok jadier. Hlavnou snahou tvorcov modelu CUDA bolo čo najviac zjednodušiť písanie programov pre GPGPU, tak aby programátor mohol ľahko zvládnuť tento problém a písať program v podobnom duchu, ako v konvenčných jazykoch, ako je napríklad jazyk C.

V porovnaní s konvenčným programom je potrebné pochopiť tri kľúčové abstrakcie:

- hierarchiu a organizáciu vlákien,
- hierarchiu pamäte (lokálna, zdieľaná, globálna, pamäť textúr a konštánt),
- synchronizáciu pomocou bariér.

Táto abstrakcia poskytuje jemnozrnný údajový paralelizmus (viď. kapitola **??**) vykonávaný v podobe vlákien, ktorý je vnorený v rámci hrubozrnného údajového alebo funkcionálneho paralelizmu. To znamená, že programátor musí byť schopný navrhnúť rozdelenie úloh na nezávisle paralelne riešiteľné podproblémy vykonávané blokmi vlákien. Tieto podúlohy musí byť možné ďalej deliť na menšie podúlohy, ktoré sú súbežne vykonávané vláknami v rámci jedného bloku.

Tento model umožňuje širokú variabilitu vykonávania podúloh, čo umožňuje spúšťať program na širokej škále rôznych GPGPU s rôznym počtom jadier procesora. Zoznam všetkých GPGPU, ktoré podporujú model CUDA je možné nájsť online [20].

Ako príklad si môžme uviesť operáciu sčítania dvoch vektorov. Sériové prevedenie tohto programu by spočívalo v postupnom iterovaní všetkými prvkami vektorov a ich sčítavaniu tak, ako je uvedené v nasledujúcom zdrojovom kóde 4.

Zdrojový kód 4: Sériový program na sčítanie dvoch n-prvkových vektorov

```
void VecAdd(int *A, int *B, int *C, int n)
{
    for(int i=0; i<n; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

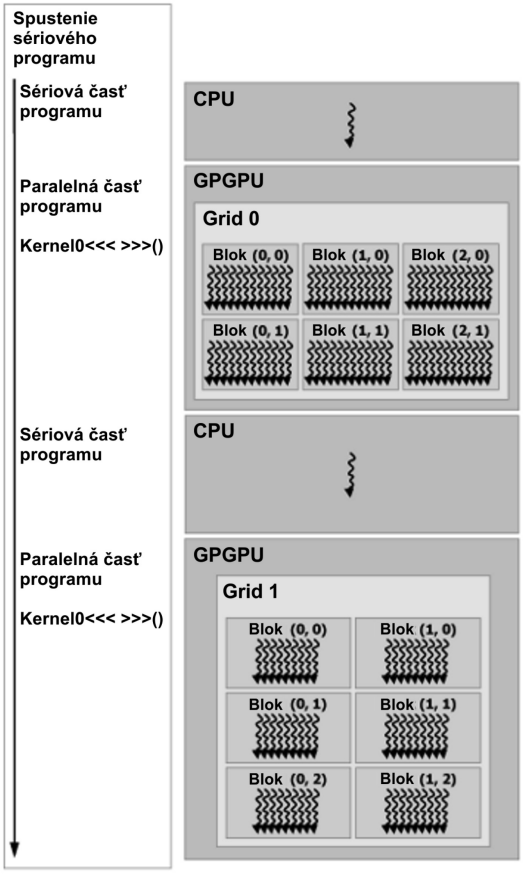
Program v CUDA sa spúšťa a začína vykonávať rovnakým spôsobom ako konvenčný program na CPU, avšak pri zavolaní špeciálnej funkcie nazývanej **kernel**, sa jej vykonávanie predá GPGPU. Takýchto kernelov môžme počas behu programu zavolať aj niekoľko. Zatiaľ, kým sú kernely vykonávané na zariadení GPGPU, zvyšok programu je vykonávaný ďalej na CPU. Na obrázku 22 je znázornený model vykonávania programu na CPU a GPGPU.

Pri volaní kernelu je zároveň potrebné špecifikovať počet vlákien a blokov, v ktorých sa má funkcia vykonávať pomocou špeciálnej syntaxe `<<< ... >>>`. Následne je možné v každom vlákne získať informáciu o poradovom čísle vlákna v rámci bloku pomocou premennej `threadIdx`. Výsledný program je uvedený v zdrojovom kóde 5.

Zdrojový kód 5: Paralelný CUDA program na sčítanie dvoch n-prvkových vektorov

```
__global__ void VecAdd(int *A, int *B, int *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    //volanie kernelu vykonávaného na GPGPU
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

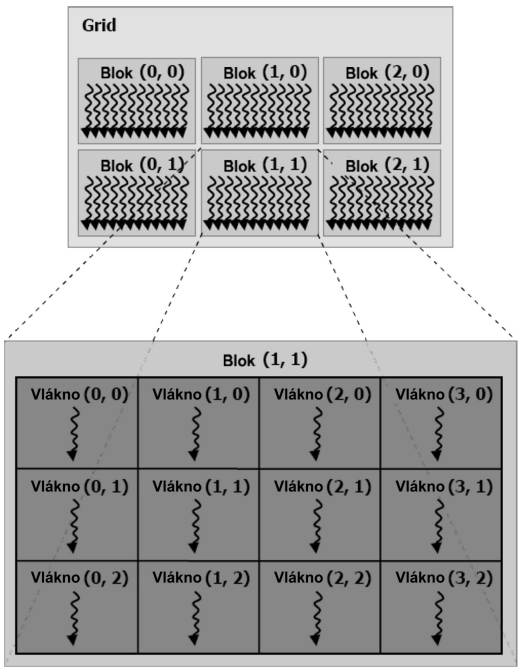


Obr. 22: Model vykonávania CUDA programu

Vlákna sú organizované v jedno-, dvoj- alebo trojdimenzionálnom priestore do väčších celkov označovaných ako **bloky vlákien**, čo umožňuje prirodzené mapovanie vlákien na jednotlivé prvky vektorov, matíc alebo priestoru. Keďže dimenzie bloku vlákien sú ohraničené, je možné ďalej bloky organizovať v jedno-, dvoj- alebo trojdimenzionálnom priestore do väčších celkov označovaných ako **mriežky blokov vlákien** (gridy). Táto hierarchia vlákien je znázornená na obrázku 23. Podstatné je uvedomiť si fakt, že všetky vlákna vykonávané súčasne v tom istom bloku sú vykonávané jedným prúdovým multiprocesorom (SM) a musia vykonávať tie isté inštrukcie. Dokonca aj v prípade použitia vetvenia je najprv vykonaná jedna vetva programu všetkými vláknami a až potom druhá vetva programu opäť všetkými vláknami, pričom sa nepotrebné výsledky zamietnu.

Poznámka: Súčasné GPGPU umožňujú používať bloky s maximálne 1024 vláknami.

Zásadným rozdielom v porovnaní s paralelným programom vykonávaným na viacjadrovom CPU je skutočnosť, že paralelne vykonávaný kernel je vykonávaný na fyzicky inom zariadení ako CPU. GPGPU z pohľadu procesora vystupuje v úlohe pomocného koprocesora. Ako CPU, tak aj GPGPU disponuje svojou vlastnou pamäťou typu DRAM, v ktorej sú uložené aktuálne spracovávané údaje. Keďže GPGPU je prispôsobená na vykonávanie veľkého množstva vlákien, hierarchia pamäte je trochu zložitejšia. Každé vlákno v bloku má k dispozícii svoju lokálnu pamäť. Každý blok vlákien má k dispozícii zdieľanú pamäť, s ktorou môže pracovať ktorékoľvek vlákno z daného bloku. Uvedené dva typy pamäte sú relatívne malé a predstavujú kapacitu radovo niekoľko kB. Ostatné údaje je možné ukladať do globálnej



Obr. 23: Hierarchia vlákien v programe CUDA

pamäte s kapacitou rádovo niekoľko stoviek MB až jednotiek GB, ku ktorej majú prístup všetky vykonávané vlákna, avšak prístup k nej je pomalší v porovnaní s lokálnou alebo zdieľanou pamäťou. Okrem toho GPGPU ešte disponuje ďalšími špecifickými typmi pamäte, ako je pamäť textúr a pamäť konštánt.

V neposlednom rade je potrebné zmieniť sa o komunikácii medzi CPU a GPGPU. Prv než je možné začať vykonávať kernel je potrebné, aby boli v pamäti GPGPU pripravené všetky potrebné údaje. Tieto je potrebné preniesť po zbernici PCIe, čo častokrát predstavuje veľké úskalie, obzvlášť v prípade častého prenosu údajov medzi pamäťou CPU a pamäťou GPGPU a späť pri častom volaní krátkych kernelov. Istým riešením je presunutie celého výpočtu na GPGPU vrátane sériových častí programu, čím sa eliminuje potreba prenášania priebežných výsledkov medzi pamäťami CPU a GPGPU po zbernici. Toto riešenie však nie je vždy najefektívnejšie. Novšie GPGPU disponujú funkcionalitou pre súčasný prenos údajov a vykonávanie kernelov, čo je možné využiť na skrátenie času čakania na pripravenie údajov tak, že kým je vykonávaný jeden kernel, do pamäte GPGPU sú priebežne nahrávané údaje potrebné pre vykonávanie ďalšej časti programu alebo výsledky predošlého kernelu sú kopírované do pamäte CPU až počas vykonávania nasledujúceho kernelu. V takto optimalizovanom programe je možné dosiahnuť nezanedbateľnú časovú úsporu [21, 22].

7 Cloudové počítanie

V posledných rokoch sa čoraz častejšie stretávame s cloudovými riešeniami ponúkanými rôznymi spoločnosťami. Hoci definícia cloudového počítania nie je jednoznačná, väčšina definícií ho definuje z

určitého špecifického pohľadu, ako napríklad obchodný model alebo technológie a virtualizácia [23]. Národný inštitút pre štandardy a technológie (NIST) definoval **cloudové počítanie** ako model pre všadeprítomný, praktický, sieťový prístup na požiadanie k zdieľanému súboru konfigurovateľných výpočtových prostriedkov (sieť, servery, úložiská, aplikácie a služby), ktoré je možné prenajať a používať v krátkom čase s minimálnymi požiadavkami na manažovanie [24].

Prostriedky poskytované v rámci cloudových služieb sú virtualizované, pričom sa používa zdieľaný hardvér, čo umožňuje jeho efektívnejšie využívanie. Medzi ďalšie výhody pre používateľa možno zaradiť to, že:

- za cloudové služby platí iba vtedy, keď ich potrebuje a využíva ich,
- sú dostupné po sieti kdekoľvek na rôznom type koncových zariadení,
- výpočtové prostriedky sú spoločné a zdieľané,
- poskytujú veľkú mieru elasticity, navýšenie alebo zníženie výkonu podľa potreby.

Myšlienka cloudového počítania, ako taká nie úplne nová, ale pramení v iných technológiách, ako sú klastrové a gridové počítanie. Vysokovýkonné počítanie a cloudové počítanie majú spoločné to, že ich úlohou je riešiť výpočtovo náročné problémy zväčša rozdelením na menšie podproblémy, ktoré môžu byť riešené paralelne na viacerých procesoroch. Zatiaľ čo hardvérovú základňu pre vysokovýkonné počítanie tvoria finančne nákladné superpočítače a masívne paralelné multiprocesorové systémy, cloudové počítanie je skôr postavené na báze lacnejšieho komoditného hardvéru, ktorý je možné v prípade poruchý alebo obnovy jednoducho nahradiť. Gridové počítanie umožňuje využívať ako superpočítače, tak aj komoditný hardvér vďaka bežným protokolom a rozhraniam a distribuovanému manažmentu a plánovaniu úloh pomocou middleware softvéru. Rozdielom však je, že pri gridovom počítaní sú prostriedky geograficky umiestnené na rôznych miestach, zatiaľ čo v cloudovom počítaní sú centralizované u ich poskytovateľa. Toto riešenie poskytuje rýchlejšie prepojenie počítačov, a preto je vhodnejšie na riešenie aplikácií s vysokými požiadavkami na vstupno-výstupné operácie, ako napríklad analýzy veľkých dát (Big Data) [25].

Základným problémom pri používaní vysokovýkonných klastrov a superpočítačov je vysoká počiatočná investícia potrebná na ich obstaranie a následná potreba ich správy, prevádzkovania a obnovy. Ďalší aspekt je efektívnosť využívania takýchto prostriedkov. Cloudové služby tieto nedostatky odstraňujú v tom zmysle, že v cloude si môžeme prenajať také výpočtové prostriedky a v takom rozsahu, ako práve potrebujeme. Taktiež náklady na ich správu a obnovu znáša poskytovateľ služieb. Efektívnosť využívania hardvéru v cloudovom počítaní je zabezpečená práve virtualizáciou a zdieľaním prostriedkov medzi viacerými používateľmi, pričom je garantovaná izolácia medzi jednotlivými nájomníkmi cloudových služieb.

Isté prepojenie medzi svetom vysokovýkonného počítania a svetom

cloudového počítania predstavuje riešenie založené na **elastickom klastri**. Tento predstavuje zjednotený model manažovania zdrojov HPC a zdrojov cloudu. Elastický klaster umožňuje využívať ako fyzické, tak aj virtuálne zdroje, pričom spĺňa kritériá kvality s ohľadom na oneskorenie pri používaní virtuálnych zdrojov [26].

Literatúra

[1] Raúl Rojas and Ulf Hashagen. *The first computers: History and architectures*. MIT press, 2002 (cited on page 1).

[2] Paul E Ceruzzi, E Paul, et al. *A history of modern computing*. MIT press, 2003 (cited on page 1).

[3] Felix Ogban, Iwara Arikpo, and Idongesit Eteng. "Von Neumann Architecture and Modern Computers." In: *Global Journal of Mathematical Sciences* 6.2 (Sept. 2007), p. 97 (cited on page 1).

[4] Peter S. Pacheco. *An Introduction to Parallel Programming*. MA USA: Morgan Kaufmann, 2011 (cited on pages 2, 8, 11).

[5] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010 (cited on page 2).

[6] Ľ. Jánošíková. *Programovanie v jazyku symbolických adries pre 32-bitové procesory Intel*. Monografie / Žilinská univerzita. Žilinská univerzita, 2000 (cited on page 2).

[7] Nihar R Mahapatra and Balakrishna Venkatrao. "The processor-memory bottleneck: problems and solutions". In: *XRDS: Crossroads, The ACM Magazine for Students* 5.3es (1999), p. 2 (cited on page 2).

[8] Stuart E Madnick et al. *Operační systémy*. SNTL, 1981 (cited on page 5).

[9] M J Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909 (cited on page 7).

[10] Michael J Flynn. "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960 (cited on page 7).

[11] Alfred Spector and David Gifford. "The space shuttle primary computer system". In: *Communications of the ACM* 27.9 (1984), pp. 872–900 (cited on page 7).

[12] Ján Kollár. *Metódy a prostriedky pre výkonné paralelné výpočty*. Košice: elfa, 2003 (cited on page 8).

[13] W Daniel Hillis. *The connection machine*. MIT press, 1989 (cited on page 8).

[14] Zbigniew J. Czech. *Introduction to parallel computing*. UK: Cambridge University Press, 2016 (cited on pages 11, 12, 14, 15).

[15] Pawel Czarnul. *Parallel Programming for Modern High Performance Computing Systems*. Chapman and Hall/CRC, 2018 (cited on page 11).

[16] *Top500 list*. https://www.top500.org/lists/2019/06/ (cited on page 12).

[17] *GNU Project*. https://www.gnu.org (cited on page 12).

[18] NVIDIA. *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. 2019 (cited on pages 18, 19).

[19] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016 (cited on page 19).

[20] NVIDIA. *CUDA GPUs*. https://developer.nvidia.com/cuda-gpus. 2019 (cited on page 21).

[21] Miroslav Melicherčík et al. "Applicability of Graphical Processing Units to Coupled Clusters Calculations". In: *Central European Symposium on Theoretical Chemistry*. 2010, p. 97 (cited on page 23).

[22] Miroslav Melicherčík and Miroslav Kováč. "Implementácia algoritmu násobenia matíc na GPGPU s optimalizáciou prenosu údajov". In: *Prírodovedec*. 2016, pp. 16–27 (cited on page 23).

[23] Luis M Vaquero et al. "A break in the clouds: towards a cloud definition". In: *ACM SIGCOMM Computer Communication Review* 39.1 (2008), pp. 50–55 (cited on page 24).

[24] Peter Mell, Tim Grance, et al. “The NIST definition of cloud computing”. In: (2011) (cited on page 24).

[25] Nelson LS Da Fonseca and Raouf Boutaba. *Cloud services, networking, and management*. John Wiley & Sons, 2015 (cited on page 24).

[26] J Škrinárová. *Elastický klaster*. Belianum, 2017 (cited on page 25).