Cache-oblivious algorithms

Cache-oblivious algorithms

- Cache-awareness
  - Algoritmus je vyladěn pro konkrétní parametry cache
  - Prakticky obtížně proveditelné - parametrů je příliš mnoho
- Cache-obliviousness
  - Víme, že cache existuje, ale neznáme její parametry
- Cache-oblivious algorithms
  - Pohled na složitost algoritmů zohledňující existenci cache
  - Algoritmy fungující z tohoto pohledu dobře

Cache-oblivious algorithms (zjednodušený pohled)

- Cache-oblivious algorithms [1999]
  - Pohled na složitost algoritmů zohledňující existenci cache
    - Počítáme přístupy k hlavní paměti (cache misses)
    - Složitost je funkcí velikosti vstupu (n) a velikosti cache (C)
      - *Zjednodušeno* (parametrem bývá i velikost cache line)
    - Zkoumá se obvykle asymptotické chování (vzhledem k n)
      - O(f(n,C))
    - Počet výpadků je shora omezen časovou složitostí O(t(n))
      - Lze popsat relativně vůči časové složitosti (pokud máme její oboustranný odhad)
      - O(f(n,C)) = O(t(n)*g(n,C))
      - g(n,C) říká, jak často algoritmus generuje cache miss
      - V mnoha případech pro velká n nezávisí na n, tj. O(g(n,C)) = O(g(C))
      - U dobrých algoritmů g(C) klesá k nule pro velká C
  - Příklad: Násobení matic (n*n)
    - Učebnicový algoritmus (i-j-k iterace)
      - Pro n^2>C každý přístup (k pravému operandu) generuje cache miss
      - Složitost $O(n^3)$ - nezávisí na C; g(C) = 1
    - Cache-oblivious algoritmus (rekurzivní dělení)
      - $O(C^{-1/2} * n^3)$ tj. $g(C) = C^{-1/2}$

- Cache-oblivious algorithms
  - Definice předpokládá pouze 1 úroveň cache
    - Cílem je však dobré chování pro všechna C (z "rozumného" intervalu)
    - Dobré chování pro cache jakékoliv velikosti implikuje dobré chování pro více úrovní
      - Fakticky zbytečně optimalizujeme i pro ty velikosti cache, které v daném systému nejsou
- Který algoritmus je lepší?
  - Minimalizujeme g(C) pro všechna C (nikoliv asymptotické chování)
    - To není jednoznačné zadání, ale porovnání obvyklých g jednoznačné bývá (např. $C^{-1/2} < 1$ pro cache-oblivious vs. textbook matrix multiplication)

Cache-oblivious algorithms

- Reálné důsledky cache-oblivious složitosti
  - Konkrétní hardware má několik úrovní cache velikostí $\{C_i\}$ s latencí $\{L_i\}$ a propustností $\{P_i\}$
  - Zdržení výpočtu kvůli latenci lze odhadnout jako

$$L(n) = t(n)*(\Sigma\ g(n,C_i)*L_i)$$

  - Pesimistický odhad předpokládající, že v době výpadku cache vše ostatní stojí
  - V reálných případech nemusí určité množství výpadků zdržovat vůbec
  - Propustnost paměťové hierarchie pro daný algoritmus:

$$T(n) = t(n)*(max\ g(n,C_i)/P_i)$$

  - P_i udává zvládnutelný počet výpadků cache i-té úrovně za jednotku času
  - $t(n)*g(n,C_i)$ je celkový počet výpadků cache i-té úrovně
  - $t(n)*g(n,C_i)/P_i$ je čas potřebný pro obsloužení tohoto počtu výpadků
  - T(n) je dolní odhad času potřebného k provedení algoritmu
  - Propustnost paměťové hierarchie hraje roli, pokud T(n) > t(n), tedy

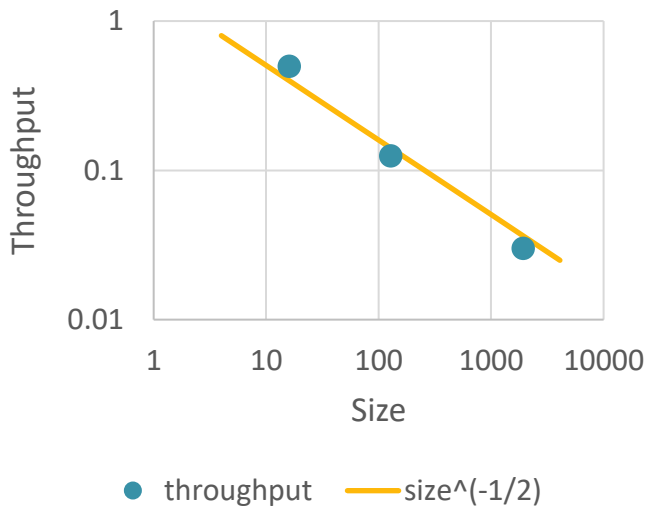
$$max\ g(n,C_i)/P_i > 1$$

  - Nerovnost má smysl pouze tehdy, pokud t(n) je udáváno ve skutečných časových jednotkách, ne pouze asymptoticky
  - I v asymptotickém případě to však dává vodítko, která úroveň cache rozhoduje

Reálné hodnoty - Intel Broadwell (Xeon E5-4655 v4)

	total size [KB]	size per thread [KB]		latency [cycles]	total throughput (lines per cycle)	throughput per thread (lines per cycle)
L1	256	**16**	L2-to-L1	12	8	**0.5**
L2	2048	**128**	L3-to-L2	~22	1	**0.125**
L3	30720	**1920**	DRAM-to-L3	~70	0.48	**0.03**

- 8 cores, 16 threads, 2.5 GHz
  - L1: 32KB per core
  - L2: 256KB per core
  - L3: 30MB shared
  - DRAM: 9.6 GT/s = 76.8 GB/s = 1.2 G lines/s
- Throughput P vs. size C
  - roughly proportional to $C^{-1/2}$
  - cf. cache-oblivious matrix multiplication



Cache-oblivious algorithms

- Stále většinou jde o asymptotický pohled vzhledem k n
  - Zanedbáváme multiplikativní konstanty, neřešíme rozdíl čtení/zápis
    - Pro reálné problémy může být výhodnější asymptoticky horší algoritmus
    - Práce s pamětí nemusí být kritické místo algoritmu
  - Zanedbané konstanty často výrazně zkreslují chování pro malá C
    - Jádro algoritmu bývá vhodnější implementovat s přibližnou znalostí chování L1 cache
    - Překladač obvykle neumí sám využít registry jako L0 cache
- Jako inspirace se vždy vyplatí

- Rozděl a panuj
  - Problém P_0 se dělí na podproblémy P_1 a P_2
  - Čas: $T_0 = T_1 + T_2$
    - Obvykle používané dělení: $T_1 = T_2 = \frac{T_0}{2}$
  - Použité adresy: $A_0 = A_1 \cup A_2$; $|A_0| \leq |A_1| + |A_2|$
    - $m(0, T_1) = |A_1|; m(T_1, T_1 + T_2) = |A_2|$
    - Není ale jasné, kolik adres použijí časové úseky zasahující do obou podproblémů
      - Typicky to bude více než uvnitř jednoho z podproblémů
      - $m\left(\frac{T_0}{2}\right) \geq \frac{|A_1|+|A_2|}{2} = \frac{|A_0|+|A_1 \cap A_2|}{2}$
    - Cíl optimalizace: Minimalizovat $m\left(\frac{T_0}{2}\right)$
      - Návod: Minimalizovat $A_1 \cap A_2$

Násobení matic - rozděl a panuj

- C = A * B, velikosti A[i,k], B[k,j], C[i,j]
  - Celková adresovaná paměť: $m(i \cdot k \cdot j) = i \cdot k + j \cdot k + i \cdot j$
- i-split - dělení problému v dimenzi i
  - $i_0 = i_1 + i_2;\ j_0 = j_1 = j_2;\ k_0 = k_1 = k_2$
  - Matice A a C se dělí napůl - podproblémy jsou na nich disjunktní
  - Matice B se nedělí - oba podproblémy ji používají celou: $|A_1 \cap A_2| = k_0 \cdot j_0$
- j-split a k-split - další možnosti dělení
  - Volí se dělení produkující nejmenší $|A_1 \cap A_2|$
  - Tomu odpovídá split podél největšího rozměru
  - Pokud $i_0 \geq j_0; i_0 \geq k_0$ a $i_1 = \frac{i_0}{2}$
    - $m(i_1 \cdot k_0 \cdot j_0) = \frac{i_0}{2} \cdot k_0 + j_0 \cdot k_0 + \frac{i_0}{2} \cdot j_0 \leq \frac{2}{3} m(i_0 \cdot k_0 \cdot j_0)$
    - $m\left(\frac{1}{2}T\right) \leq \frac{2}{3} m(T)$
  - Pro čtvercové matice trojice i,j,k-split vede k výsledku
    - $m\left(\frac{1}{8}T\right) = \frac{1}{4} m(T)$

Násobení matic - rozděl a panuj

- C = A * B, velikosti A[i,k], B[k,j], C[i,j]
  - Celková adresovaná paměť: $m(i \cdot k \cdot j) = i \cdot k + j \cdot k + i \cdot j$
  - Pro čtvercové matice trojice i,j,k-split vede k výsledku
    - $m\left(\frac{1}{8}T\right) = \frac{1}{4}m(T)$
    - $m(w) = q \cdot w^{\frac{2}{3}}$ pro konstantu $q \sim 3$
    - $m^{-1}(C) = (\frac{C}{q})^{\frac{3}{2}}$
    - $\frac{\partial m}{\partial w}(w) = \frac{2}{3}q \cdot w^{-\frac{1}{3}}$
    - frekvence výpadků $g(C) = \frac{\partial m}{\partial w}(m^{-1}(C)) = \frac{2}{3}q^{\frac{3}{2}} \cdot C^{-\frac{1}{2}}$

Cache-oblivious algorithms – vliv bloků

- Zjednodušená složitost uvažuje pouze velikost cache C
  - Výsledek nezávisí na způsobu uložení dat v pamětí
- Úplná cache-aware složitost uvažuje i velikost bloku B
    - Blokem je řádka cache případně stránka vzhledem k TLB
    - Velikost C se udává v blocích
  - f(C,B)
  - Počítají se přesuny bloků mezi cache a hlavní pamětí
    - Lepší paměťová struktura jich spotřebuje méně

Cache-oblivious algorithms – vliv bloků

- Příklad: Násobení matic (n*n)
  - Učebnicový algoritmus (i-j-k iterace), uložení po řádcích
    - Pro n>C každý přístup (k pravému operandu) generuje cache miss
    - $f(C,B) = O(n^3)$ - nezávisí na C; $g(C) = 1$
    - Při uložení po sloupcích zdržuje přístup k levému operandu
    - Při uložení po čtvercích $B^{1/2} * B^{1/2}$ se složitost zlepší na $O(B^{-1/2} * n^3)$
  - Cache-oblivious algoritmus (rekurzivní dělení, rekurzivní uložení)
    - $O(C^{-1/2} * B^{-1} * n^3)$ tj. $g(C) = C^{-1/2} * B^{-1}$
    - Typické hodnoty konstant (neřešíme společnou multiplikativní konstantu)
      - 8 registrů, double: C = 8, B = 1, g(C) = 1/2.8, jednotková cena 1/3 cyklu CPU
      - 8 SSE registrů, double: C = 16, B = 2, g(C) = 1/8
      - 32KB L1 cache, double: C = 4K, B = 8, g(C) = 1/512, jednotková cena 1 cyklus CPU
      - 64-entry TLB, double: C = 64, B = 512, g(C) = 1/4K
      - 8MB L3 cache, double: C = 1M, B = 8, g(C) = 1/8K, jednotková cena 8 cyklů CPU
      - 8 GB RAM, 64 KB blok, double: C = 1G, B = 8K, g(C) = 1/256M, jednotka 300K cyklů (SSD)
      - 512 GB SSD, 64 KB blok, double: C = 64G, B = 8K, g(C) = 1/4G, jednotka cca 2M cyklů (HDD)

Nejvýznamnější cache-oblivious algoritmy

- Násobení čtvercových matic
  - $O(1 + \frac{n^2}{B} + \frac{n^3}{B\sqrt{C}})$
  - Strassen: $n^{\log 2(7)}$
- Transpozice matice
  - $O(1 + \frac{mn}{B})$
- FFT
  - $O(1 + \frac{n}{B}(1 + \frac{\log(n)}{\log(C)}))$
- Funnelsort
  - $O(1 + \frac{n}{B}(1 + \frac{\log(n)}{\log(C)}))$
- Binární vyhledávací stromy (van Emde Boas)
  - $O(\frac{\log(n)}{\log(B)})$ pro $C \ll n$

Nejvýznamnější cache-oblivious algoritmy

- Binární vyhledávací stromy
  - Implementace s ukazateli
    - Obvykle náhodné rozmístění v paměti
      - Čtení ukazatelů obvykle nevyvolává další výpadky díky prostorové lokalitě jednoho uzlu
      - Při opakovaném vyhledávání je prvních $\log(C)$ pater stromu přítomno v cache
      - Počet cache miss: $O(\log(n) - \log(C))$
  - Serializace stromu do pole
    - Kořen uprostřed – binární vyhledávání v setříděném poli
      - Prostorová lokalita v $\log(B)$ posledních patrech stromu ušetří
      - $O(\log(n) - \log(C) - \log(B))$
    - Kořen vlevo – viz heapsort
      - Prostorová lokalita v prvních patrech stromu zlepšuje efektivitu cache
      - $O(\log(n) - \log(CB)) = O(\log(n) - \log(C) - \log(B))$
    - van Emde Boas (1975) – rekurzivní dělení napůl vzhledem k výšce stromu
      - Původní účel struktury: Časová složitost search/insert/delete $O(\log m)$ pro m-bitové klíče
      - Strom o n uzlech se realizuje jako strom o $\sqrt{n+1} - 1$ uzlech, na jehož listy bude navázáno $\sqrt{n+1}$ dalších stromů po $\sqrt{n+1} - 1$ uzlech
      - Rekurze odpovídá obecnému vzoru "dělte čas napůl"
      - Počet cache miss: $O(\frac{\log(n) - \log(C)}{\log(B)})$

Cache-oblivious algoritmy

- Otázky spojené s cache-oblivious algoritmy
  - Ignorujeme strategii výměny cache
    - Nevadí: LRU se nechová hůře než ideální cache poloviční velikosti
  - Ignorujeme vícevrstevnost cache hierarchie
    - Nevadí, pro inkluzivní cache se každá vrstva chová stejně, jako by byla samostatná
  - Ignorujeme nedokonalou asociativitu cache
    - Teoreticky: Náhrada cache hashovací tabulkou s dobrou hashovací funkcí složitost nezhorší
    - Prakticky: Nedokonalost hashovací funkce vadí
      - Často jde o triviální funkce vyřezávající bity z adresy