Sekvenční architektury - paměť

Tomáš Oberhuber
tomas.oberhuber@fjfi.cvut.cz

11. února 2024

Videa na Youtube:

První část

Druhá část

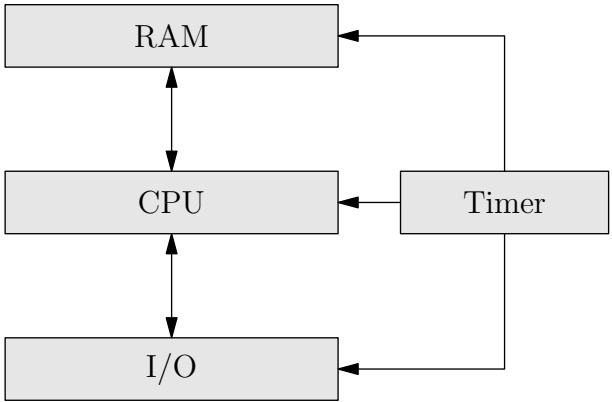
Sekvenční architektury

Definition

Za **sekvenční systém** budeme považovat systém založený na von Neumannově architektuře s **jednou** centrální jednotkou - CPU.

Sekvenční systém může být doplněn o **aritmeticko logickou jednotku** - ALU.

Von Neumannova architektura - připomenutí

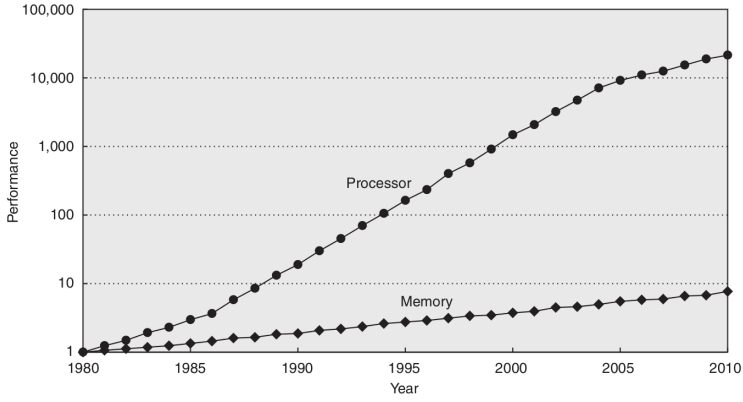


Jak urychlit výpočet na sekvenčním systému?

Jak urychlit výpočet na sekvenčním systému?

- na straně hardware
  - zrychlit spojení mezi CPU a RAM
  - zvýšit výkon CPU
- na straně software
  - optimalizovat přístup do paměti
  - eliminovat zbytečné instrukce programu nebo instrukce náročné na provedení

Paměťový subsystém PC



Zdroj: Henessy, Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 2011.

Paměťový subsystém PC

Výkon CPU roste mnohem rychleji než rychlost RAM:

- *memory wall*
- *Von Neumann bottleneck*

Rychlost komunikace mezi pamětí a procesorem je dána:

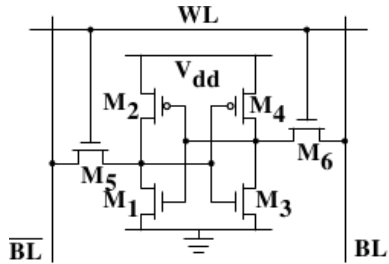
- **latencí** (latency)
  - udává, za jak dlouho je paměťový modul schopný vyhledat požadovaná data
  - je podstatná tehdy, pokud čteme malé bloky dat z různých míst v paměti
- **přenosovou rychlostí** (bandwidth)
  - udává, jak rychle lze přenášet bloky dat po jejich nalezení
  - uplatní se při práci s velkými souvislými bloky dat

Typy pamětí

Rozlišujeme dva základní typy pamětí:

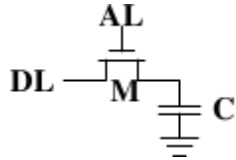
- statická paměť - *static RAM*
- dynamická paměť - *dynamic RAM*

Statická paměť



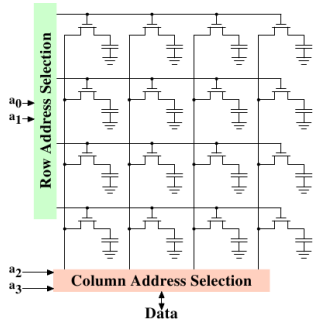
- jeden bit je reprezentován 6 tranzistory
- tato paměť je velmi rychlá a nevyžaduje žádné obnovovací cykly
- využívá se zejména pro cache procesorů
- je velmi drahá

Dynamická paměť



- jeden bit je reprezentován 1 tranzistorem a 1 kondenzátorem
- tato paměť je jednodušší, ale vyžaduje obnovovací cykly pro dobíjení kondenzátoru
- kondenzátory se vybíjejí při čtení, ale i samovolně
- během obnovovacího cyklu nelze s pamětí pracovat
- tento typ paměti je ale výrazně menší (a tedy levnější) než statická paměť a má menší spotřebu energie

Organizace dynamické paměti



- DRAM nelze organizovat lineárně tj. připojit každou paměťovou buňku k CPU přímo
  - při 4 Gb RAM by to vyžadovalo 2^{32} spojů
- adresa buňky se proto zakóduje jako binární číslo a je pak zpracována demultiplexorem
- aby nemusel být demultiplexor příliš složitý, organizuje se DRAM do 2D pole

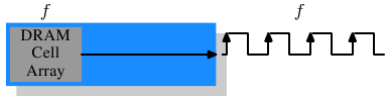
Přístup do DRAM

- celá adresa se rozdělí na index řádku (RAS) a index sloupce (CAS)
- paměťový modul nejprve přijme RAS a "připojí" požadovaný řádek
- následně přijme CAS a "připojí" požadovanou buňku
- pokud pracujeme s několika buňkami v jednom řádku, lze k nim přistupovat jen pomocí poslání jednoho RAS a několika CAS
- to může být výrazně rychlejší
- **je lepší zpracovávat data sekvenčně nebo v malých blocích, než náhodně přistupovat na různá místa v paměti**

Typy pamětí DRAM

Nejčastěji se používá synchronní DRAM tj. SDRAM.

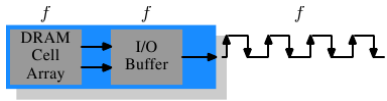
- nejjednodušší typ je SDR SDRAM (*Single Data Rate SDRAM*)



- paměťový modul a sběrnice jsou taktovány stejně tj. 100-266 MHz.
- při 100 MHz je datová propustnost $100,000,000 \times 64b/s = 800MB/s$.

Typy pamětí DRAM

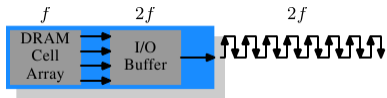
Nástupce pamětí SDR SDRAM je DDR SDRAM (*Double Data Rate SDRAM*)



- na rozdíl od SDR SDRAM přenáší data i na sestupné hraně signálu
- aby bylo co přenášet, obsahuje paměťový modul vyrovnávací paměť, která je připojena ke dvěma paměťovým čipům
- při 100 MHz je datová propustnost $2 \times 100 \times 64 Mb/s = 1,600 MB/s$.
- zvýšila se tím datová propustnost, aniž by se navýšil takt paměťových modulů, což by bylo energeticky velmi náročné

Typy pamětí DRAM

DDR2 SDRAM - zvyšuje frekvenci přenosu dat na dvojnásobek



- paměťové čipy jsou taktovány stejně, ale řadič pracuje rychleji
- proto je potřeba číst v jednom taktu z celkem 4 čipů
- při 100 MHz je datová propustnost $4 \times 100,000,000 \times 64b/s = 3,200MB/s$
- při 266 MHz je datová propustnost $4 \times 266,000,000 \times 64b/s = 8,512MB/s$

Typy pamětí DRAM

DDR3 SDRAM - zvyšuje frekvenci přenosu dat na čtyřnásobek

- při 100 MHz je datová propustnost $8 \times 100,000,000 \times 64b/s = 6,400MB/s$
- při 266 MHz je datová propustnost $8 \times 266,000,000 \times 64b/s = 17,024MB/s$

DDR4 SDRAM - navyšují takty paměťových modulů (dostupné od roku 2014)

- při 200 MHz je datová propustnost $8 \times 200,000,000 \times 64b/s = 12,800MB/s$
- při 300 MHz je datová propustnost $8 \times 300,000,000 \times 64b/s = 19,200MB/s$
- při 400 MHz je datová propustnost $8 \times 400,000,000 \times 64b/s = 25,600MB/s$

DDR5 SDRAM - byl schválen v roce 2020.

- opět zdvojnásobuje datovou propustnost
- snižuje energetickou náročnost

Typy pamětí DRAM

- viděli jsme, jak dochází k urychlení přenosu dat mezi CPU a RAM
- bylo ho dosaženo jen díky paralelnímu přístupu do více paměťových modulů
- z modulu se čte blok 64 bitů = 8 bytů najednou
- pokud program ale využije jen jeden byte, není datová sběrnice využita plně efektivně
- podobný trik využívá i technologie Dual/Triple/Quad channel
  - data se ukládají do dvou,tří nebo čtyř paměťových modulů na přeskáčku
  - DDR5 quad channel $\Rightarrow 4 \times 67.2 GB/s = 268.8 GB/s$

Žádná z jmenovaných technologii zatím výrazně nesnížila latenci.

Cache

Za tím účelem se používají malé a rychlé vyrovnávací pamětí postavené na čipech SRAM - **cache**

- jde o poměrně malou ale velice rychlou paměť
  - velikost je řádově $1/1000$ velikosti operační paměti
  - přístup do RAM je cca. 200 cyklů CPU
  - přístup do cache je 3-15 cyklů CPU
- je implementována přímo na stejném čipu jako procesor a běží na stejném taktu
- v cache se udržuje kopie některých dat z operační paměti, pokud se pracuje s těmito daty, CPU sahá přímo do cache a ne do pomalé operační paměti
- využívá se faktu, že u operační paměti je snazší zvýšit přenosovou rychlost, než zkrátit latenci
- do cache se načítají větší souvislé bloky dat, ke kterým je pak rychlejší přístup
- správa cache je plně v režii CPU, programátor nebo OS mají jen malou možnost ovlivnit cache

Cache

- současné procesory obsahují několik úrovní cache
- označují se L1, L2 a L3
- L1 se dělí na datovou a instrukční
- čím menší číslo, tím menší ale rychlejší cache je

Cache

CPU se snaží, aby cache obsahovala ta data, se kterými program právě pracuje.

- procesor předpokládá, že právě běžící program pracuje s menšími ucelenými bloky dat a nepřistupuje do operační paměti na zcela náhodné adresy
  - tomu se říká prostorová lokalita - *spatial locality*
- dále se předpokládá, že program nejprve zpracuje jeden blok dat, pak se posune k dalšímu bloku a k tomu původnímu se nevrací
  - to je časová lokalita - *temporal locality*

Cache

- procesor provádí tzv. *prefetching*, tj. načítání dat s předstihem
- pokouší se odhadnout, jaká data budou brzy potřeba a načíst si je do cache dřív, než si o to řekne program sám
- pak je schopen k nim poskytnout mnohem rychlejší přístup

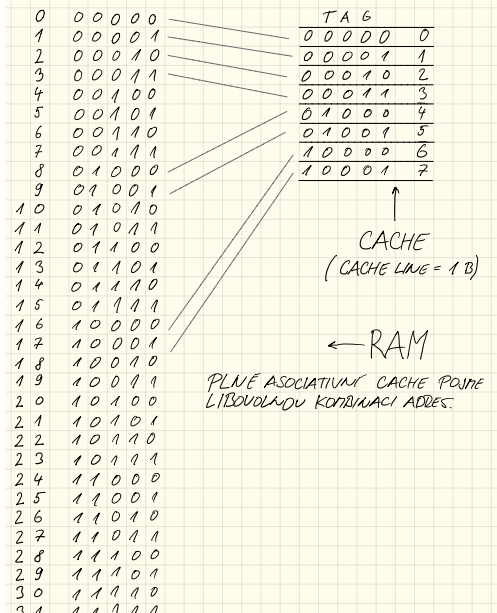
Jak funguje cache?

- data se do cache přenáší ve větších blocích, nejčastěji 64 bytů, což se provede v celkem 8 přenosech
- SDRAM jsou pro tento přenos optimalizovány a ušetří se tím vysílaní RAS a CAS signálů
- bloku dat, který je přenesen do cache najednou se říká *cache line*
- každá cache line musí být označena podle své adresy v operační paměti
- pokud chce program pracovat s určitou adresou, musí procesor nejprve zjistit, zda má tato data kešovaná nebo ne
- to musí být pochopitelně velmi rychlé

Plně asociativní cache

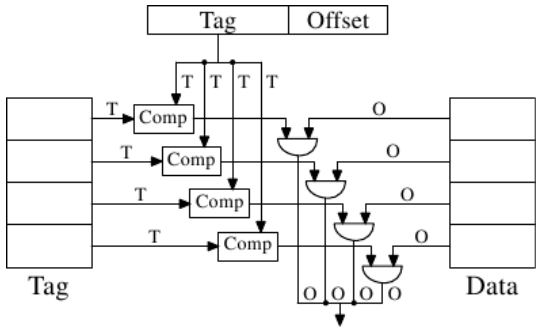
Nejjednodušší přístup k organizování cache je *fully associative cache* ...

Plně asociativní cache



Plně asociativní cache

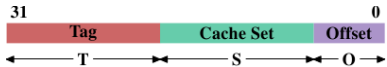
- u plně asociativní cache přísluší každé cache line jeden komparátor, který porovná tag uložené cache line s cache line, která je vyžadovaná od CPU



Plně asociativní cache

Na systémech s 32-bitovým adresováním se ...

- celá adresa se rozdělí na tag a offset
- offset odpovídá adrese v rámci cache line, tj. má velikost $O = 6$ bitů
- zbývající část $T = 26$ bitů připadne tagu



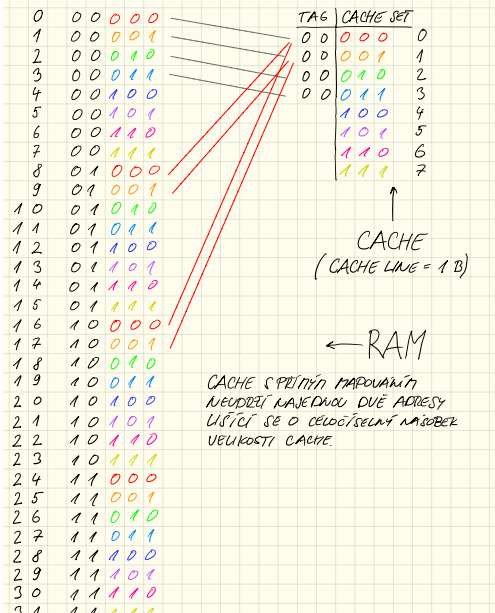
Obrázek: $T = 26$, $S = 0$ a $O = 6$

Plně asociativní cache

- jednotlivé cache line mohou odpovídat libovolným místům v paměti
- při velikosti cache 4 MB, máme celkem $4M/64 = 65,536$ komparátorů
- to by bylo velice náročné z hardwarového pohledu a drahé na výrobu

Další možností je zkonstruovat cache s přímým mapováním – *direct mapped cache*...

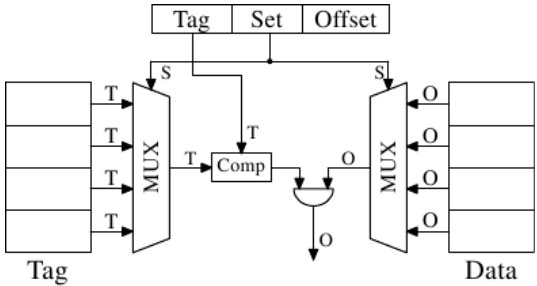
Cache s přímým mapováním



Cache s přímým mapováním

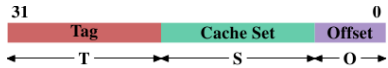
- pokud CPU potřebuje vyhledat data v cache, podívá se na část adresy odpovídající cache set
- okamžitě ví, kam má do cache sáhnout (to provádí multiplexor)
- aby zjistil, zda je zde uložena požadovaná cache line, provede porovnání těch částí adresy, které odpovídají tagu
- na to stačí jeden komparátor

Cache s přímým mapováním



Cache s přímým mapováním

- mějme opět 4 MB cache tj. $65,536$ cache line
- celou operační paměť si můžeme rozdělit právě na $65,536$ segmentů, kterým se říká cache set
- každou cache set lze adresovat pomocí 16 bitů
- spolu s offsetem to dělá $16 + 6 = 22$ bitů
- zbylých 10 bitů zůstává pro tag



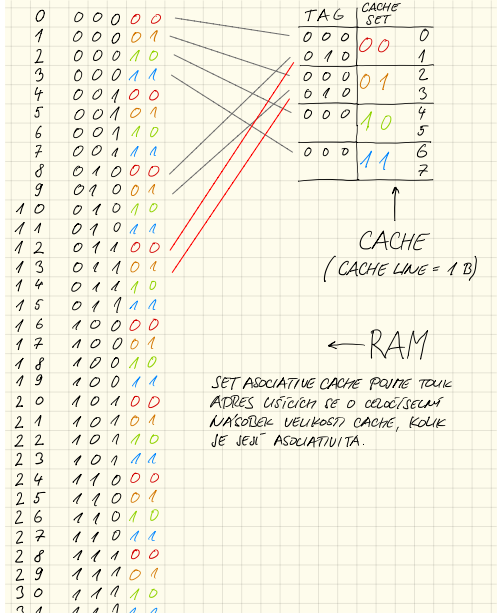
Obrázek: $T = 10$, $S = 16$ a $O = 6$

Cache s přímým mapováním

- tento typ cache je mnohem jednodušší na konstrukci
- problém je, že nelze mít v cache dvě cache line, které se liší jen tagem
- takové adresy se opakují po $2^{16+6} = 4M$, tj. velikost cache
- to sice dobře souhlasí s podmínkou spatial locality, ale jakmile aplikace pracuje s větším blokem paměti, je tento přístup neefektivní

V praxi se volí kombinace obou přístupů, tj. *set associative cache*...

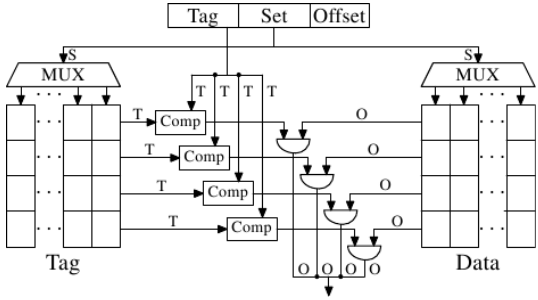
Set associative cache



Set associative cache

- celou cache rozdělíme na menší počet cache sets, ale do každé z nich umožníme uložit větší počet cache line – 2, 4 nebo 8
- velikost cache set adresy se zmenší o $1, 2$ nebo 3 bity a tagová část se příslušně zvětší
- multiplexor vybere příslušnou cache set
- v ní je nyní 2 až 8 cache lines
- pomocí 2 až 8 komparátorů se provede porovnání tagové adresy

Set associative cache



Set associative cache

L2 Cache Size	Associativity							
	Direct		2		4		8	
	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64
512k	27,794,595	20,422,527	25,222,611	18,303,581	24,096,510	17,356,121	23,666,929	17,029,334
1M	19,007,315	13,903,854	16,566,738	12,127,174	15,537,500	11,436,705	15,162,895	11,233,896
2M	12,230,962	8,801,403	9,081,881	6,491,011	7,878,601	5,675,181	7,391,389	5,382,064
4M	7,749,986	5,427,836	4,736,187	3,159,507	3,788,122	2,418,898	3,430,713	2,125,103
8M	4,731,904	3,209,693	2,690,498	1,602,957	2,207,655	1,228,190	2,111,075	1,155,847
16M	2,620,587	1,528,592	1,958,293	1,089,580	1,704,878	883,530	1,671,541	862,324

Tabulka: Efekt velikosti cache, asociativity a cache line na počet "cache misses".
Testováno na gcc.

V linuxu lze asociativitu cache zjistit příkazem:

```
cat /sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity
```

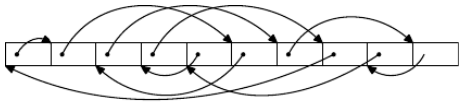
Testy přístupů do paměti

Příklad:

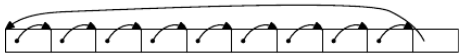
Budeme procházet následující spojový seznam:

```
1  template< int Size >
2  class ArrayElement
3  {
4     ArrayElement* next;
5     long int data[ Size ];
6  }
```

- všechny prvky seznamu se alokují jako velké pole
- následně se propojí buď sekvenčně nebo náhodně



Random



Sequential

Testy přístupů do paměti

- seznam budeme pouze **opakovaně** procházet a budeme načítat první položku v poli `data`
- hodnota `Size` simuluje různou velikost jednoho prvku seznamu
- `long int` má stejnou velikost jako ukazatel, tj. 32 nebo 64 bitů podle toho, jestli máme 32 nebo 64 bitový systém

Testy přístupů do paměti

Testy budeme provádět na:

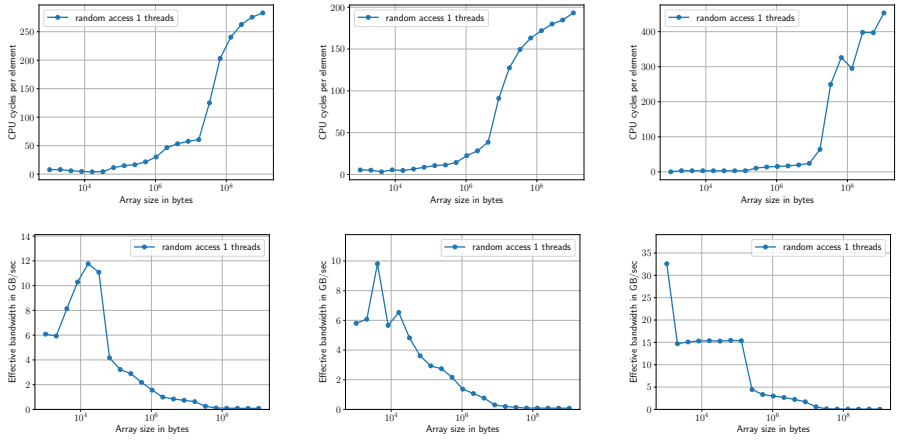
- Intel Xeon 6134 (2017)
  - 8 jader
  - L1 cache - 32 kB
  - L2 cache - 1024 kB
  - L3 cache - 24.75 MB
  - paměti DDR4
- AMD Epyc 7281 (2017)
  - 16 jader
  - L1 cache - 32 kB
  - L2 cache - 512 kB
  - L3 cache - 4 MB
  - paměti DDR4 - 170 GB/s na socket
- Apple M1 Pro (2022)
  - 8 výkonných jader
    - L1 cache - 192 kB instrukční a 128 kB datová
    - L2 cache - 12 MB sdílená
  - 2 efektivní jádra
    - L1 cache - 128 kB instrukční a 64 kB datová
    - L2 cache - 4 MB sdílená
  - paměti Low Power DDR5

Náhodné přístupy

Test pro náhodné přístupy do paměti spustíme skriptem:

```
paa-examples/sequential-architectures/memory-access/run-random-test
```

Náhodné přístupy



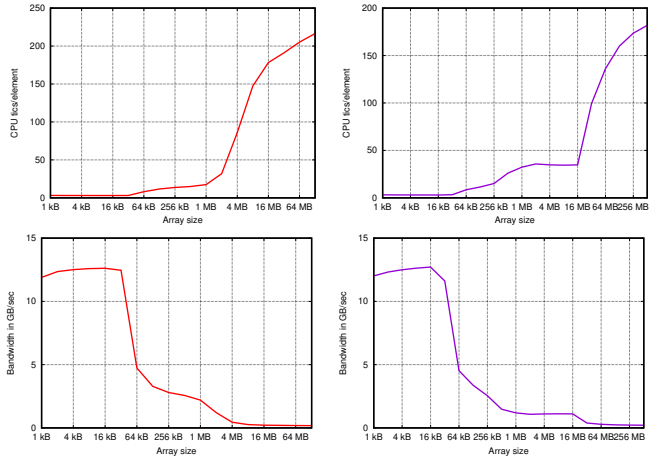
Obrázek: Výsledky s náhodným procházením pole na Intel Xeon Gold 6134 (vlevo), AMD Epyc 7281 (uprostřed) a Apple M1 (vpravo).

Náhodné přístupy

Rychlost náhodných přístupů tak přibližně vychází na:

- RAM $\approx$ 200-400 cyklů
- L2 cache $\approx$ 10-20 cyklů
- L1 cache $\approx$ 3-4 cykly

Náhodné přístupy



Obrázek: Výsledky s náhodným procházením pole na Intel Core 2 s 16Kb L1 cache, 4MB L2 cache a pamětmi DDR2 (vlevo) a Intel Xeon E5 2630 s 32Kb L1 cache, 256Kb L2 cache a 20 MB L3 cache

Sekvenční přístupy

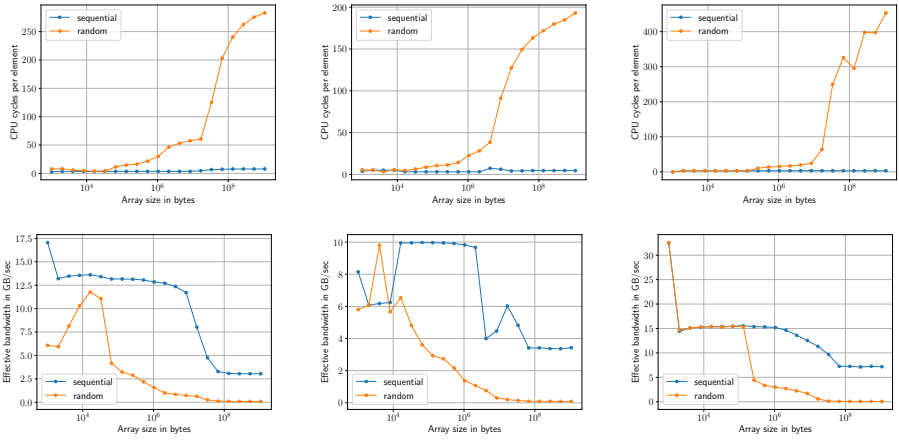
Test pro náhodné přístupy do paměti spustíme skriptem:

```
paa-examples/sequential-architectures/memory-access/run-sequential-test
```

Sekvenční přístupy

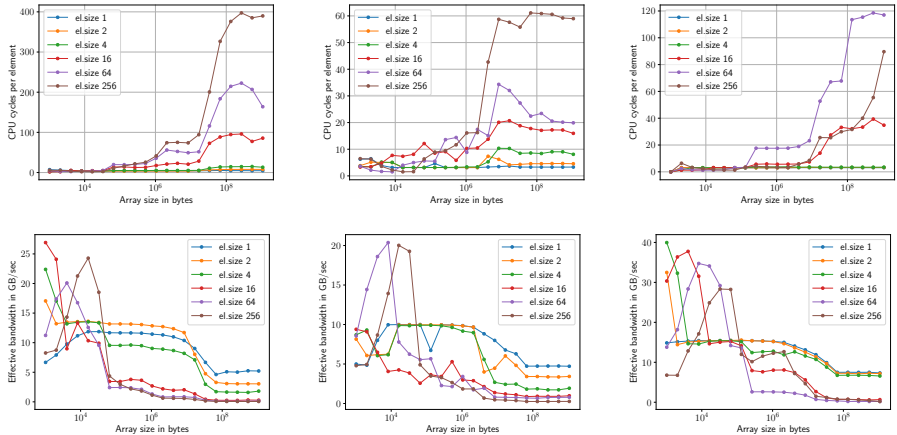
- procesor provádí prefetching, tj. načítá několik cache line dopředu
- je-li `Size = 1`, do jedné cache line se vejde 8 prvků pole
- je-li `Size = 8` je to jen 1
- je-li `Size = 32` je jeden element čtyřikrát větší než cache line

Sekvenční přístup



Obrázek: Porovnání náhodné a sekvenční procházení pole na Intel XEON Gold 6134 (vlevo), AMD Epyc 7281 (uprostřed) a Apple M1 Pro (vpravo).

Sekvenční přístupy

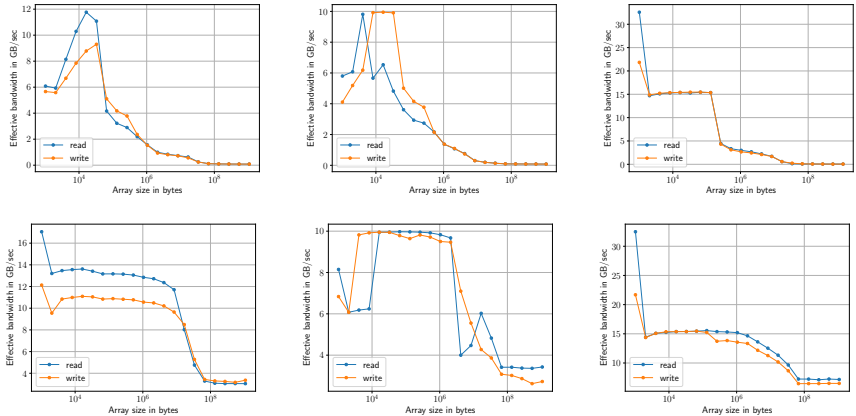


Obrázek: Výsledky se sekvenčním procházením pole s různou velikosti `Size` na Intel XEON Gold 6134 (vlevo), AMD Epyc 7281 (uprostřed) a Apple M1 Pro (vpravo).

Zapisování dat

- pokud je nějaká cache line změněna, jsou dvě možnosti, co udělat
  - provést okamžitý zápis do RAM – *write-through cache*
    - je to jednoduchá strategie, ale na příliš účinná
    - pokud program opakovaně zapisuje do stejné proměnné, je většina zápisů do operační paměti zbytečná
  - provést zápis do RAM později – *write-back cache*
    - cache line se označí jako *dirty*
    - ve chvíli, kdy je potřeba uvolnit nějaká data z cache, kontroluje se zda cache line není takto označena a pokud ano, provede se zápis do RAM

Zapisování dat



Obrázek: Porovnání procházení pole náhodně (nahoře) a sekvenčně (dole) se čtením a zápisem na Intel Xeon Gold 6134 (vlevo) AMD Epyc 7281 (oprostřed) a Apple M1 Pro (vrpavo).

Zapisování dat

Testy nám ukazují, že:

- vždy se vyplatí využívat maximálně data, která máme již v cache
- **sekvenční přístup do paměti je mnohem rychlejší než náhodný**
  - procházení pole je mnohem rychlejší, než procházení spojového seznamu
- při procházení pole s rostoucí velikostí `ArrayElement` klesá efektivita, protože využíváme jen některá data načtená do cache

Pokud to jde:

- **ukládáme data do paměti sekvenčně organizovaná tak, jak k nim nejčastěji přistupujeme**
  - to je důsledek spatial locality a prefetchingu
- **pokud s daty pracujeme během výpočtu opakovaně, je lepší rozdělit výpočet na menší bloky, které se vejdou do cache**
  - to je důsledek temporal locality

Zapisování dat

Příklad:

Může být výhodné nahradit pole struktur

```
1  struct Element
2  {
3     SmallKey key;
4     BigData data;
5  }
```

za pole struktur

```
1  struct Element
2  {
3     SmallKey key;
4     BigData* data;
5  }
```

Při procházení pole a vyhledávání podle klíče se nebude cache zahlcovat samotnými daty, která nás většinou nezajímají.

Sčítání matic

Příklad:

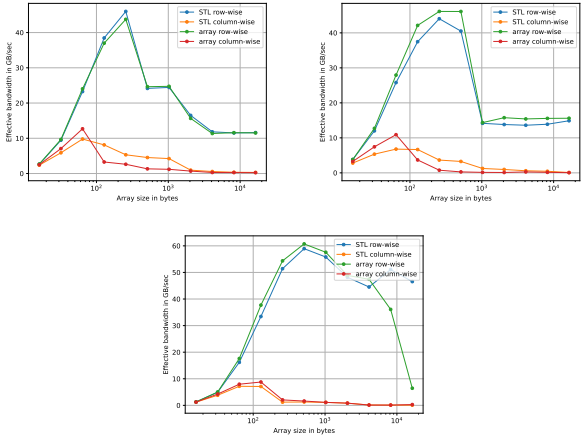
Program `matrix-addition` testuje sčítání matic. Matice jsou v paměti uloženy po řádcích. Jsou dvě možnosti jak matice sčítat:

```
1  void addMatrixRowWise( const ArrayMatrix< Real >& m )
2  {
3     for( int i = 0; i < size; i++ )
4        for( int j = 0; j < size; j++ )
5           this->matrix[ i * size + j ] +=
6              m.matrix[ i * size + j ];
7  }
```

```
1  void addMatrixColumnWise( const ArrayMatrix< Real >& m )
2  {
3     for( int j = 0; j < size; j++ )
4        for( int i = 0; i < size; i++ )
5           this->matrix[ i * size + j ] +=
6              m.matrix[ i * size + j ];
7  }
```

Pro zajímavost uděláme to samé s matici uloženou pomocí kontejneru `vector` ze standardní knihovny STD.

Sčítání matic



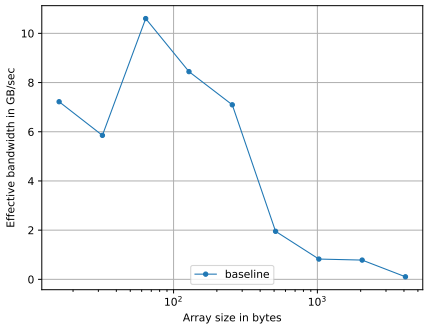
Obrázek: Porovnání sčítání matic s přístupem po řádcích a po sloupcích, uloženou jako pole nebo pomocí STD třídy `vector` na na Intel Xeon Gold 6134 (vlevo), AMD Epyc 7281 (vpravo) a Apple M1 Pro (dole).

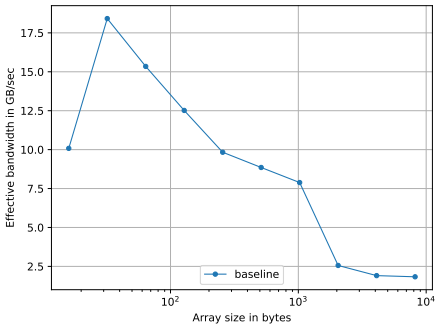
Násobení matic

Příklad:
Při násobení matic se ale přístupu po sloupcích nevyhneme.

```
1  void multiplyMatrices( const ArrayMatrix< Real >& m1,
2                         const ArrayMatrix< Real >& m2 )
3  {
4     for( int i = 0; i < size; i++ )
5        for( int j = 0; j < size; j++ )
6        {
7           Real aux( 0.0 );
8           for( int k = 0; k < size; k++ )
9              aux += m1.matrix[ i * size + k ] *
10                    m2.matrix[ k * size + j ];
11          this->matrix[ i * size + j ] = aux;
12       }
13 }
```

Násobení matic





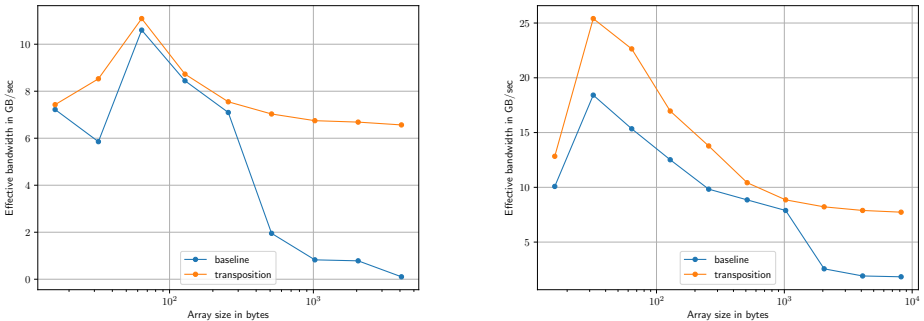
Obrázek: Klasický algoritmus pro násobení matic na AMD Epyc 7281 (vlevo) a Apple M1 Pro (vpravo).

Násobení matic

Pomůžeme si tím, že druhou matici si nejprve transponujeme:

```
1  void multiplyMatricesWithTransposition( const ArrayMatrix< Real >& m1,
2                                          const ArrayMatrix< Real >& m2 )
3  {
4     Real* transposedMatrix = new Real[ size * size ];
5     for( int i = 0; i < size; i++ )
6        for( int j = 0; j < size; j++ )
7           transposedMatrix[ i * size + j] =
8              m2.matrix[ j * size + i ];
9
10    for( int i = 0; i < size; i++ )
11       for( int j = 0; j < size; j++ )
12       {
13          Real aux( 0.0 );
14          for( int k = 0; k < size; k++ )
15             aux += m1.matrix[ i * size + k ] *
16                    transposedMatrix[ j * size + k ];
17          this->matrix[ i * size + j ] = aux;
18       }
19
20    delete[] transposedMatrix;
21 }
```

Násobení matic



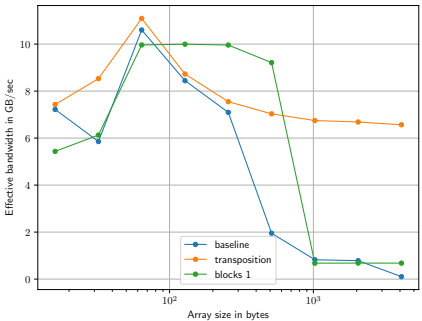
Obrázek: Klasický algoritmus pro násobení matic a algoritmus s transpozicí na AMD Epyc 7281 (vlevo) a Apple M1 Pro (vpravo).

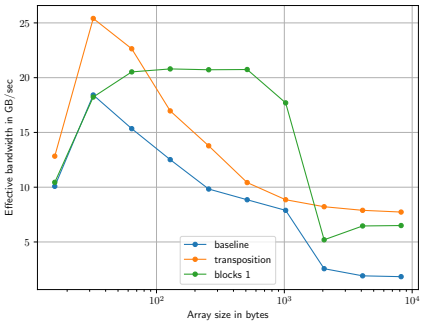
Násobení matic

- provádět transpozici ale může výrazně zvýšit nároky na paměť
- jinou možností je provést násobení blokově

```
template< int blockSize = 16 >
void multiplyMatricesBlockwise1( const ArrayMatrix< Real >& m1,
                                 const ArrayMatrix< Real >& m2 )
{
   memset( this->matrix, 0, size * size * sizeof( Real  ) );
   for( int i = 0; i < size; i += blockSize )
      for( int j = 0; j < size; j += blockSize )
         for( int k = 0; k < size; k += blockSize )
         {
            for( int i1 = i; i1 < i + blockSize; i1++ )
               for( int j1 = j; j1 < j + blockSize; j1++ )
                  for( int k1 = k; k1 < k + blockSize; k1++ )
                     this->matrix[ i1*size + j1 ] +=
                        m1.matrix[ i1*size + k1 ]*
                           m2.matrix[ k1*size  + j1 ];
         }
      }
}
```

Násobení matic





Obrázek: Klasický algoritmus pro násobení matic, algoritmus s transpozicí a blokový algoritmus na AMD Epyc 7281 (vlevo) a Apple M1 Pro (vpravo).

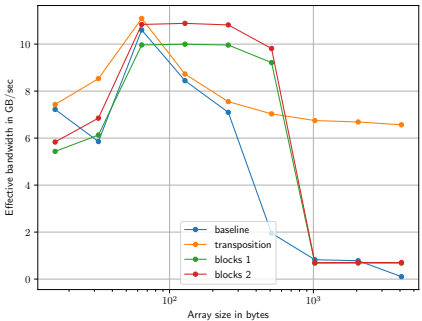
Násobení matic

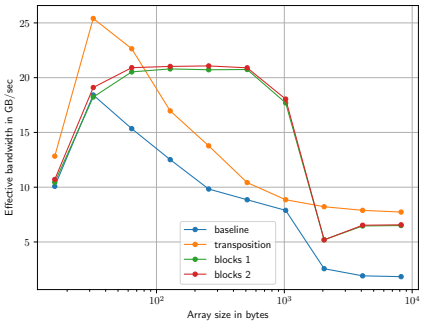
- výkon je ale překvapivě dost slabý
- důvodem je náročná aritmetika s ukazateli na řádcích 13, 14 a 15 – provedeme optimalizaci

Násobení matic

```
1  template< int blockSize = 16 >
2  void multiplyMatricesBlockwise2( const ArrayMatrix< Real >& m1,
3                                   const ArrayMatrix< Real >& m2 )
4  {
5     memset( this->matrix, 0, size * size * sizeof( Real  ) );
6     for( int i = 0; i < size; i += blockSize )
7        for( int j = 0; j < size; j += blockSize )
8           for( int k = 0; k < size; k += blockSize )
9           {
10             Real* res = &this->matrix[ i * size + j ];
11             Real* _m1 = &m1.matrix[ i*size + k ];
12             for( int i1 = 0;
13                  i1 < blockSize;
14                  i1++, res += size, _m1 += size )
15                for( int j1 = 0; j1 < blockSize; j1++ )
16                {
17                   Real* _m2 = &m2.matrix[ k*size + j ];
18                   for( int k1 = 0;
19                        k1 < blockSize;
20                        k1++, _m2 += size )
21                      res[ j1 ] += _m1[ k1 ] * _m2[ j1 ];
22                }
23          }
24 }
```

Násobení matic





Obrázek: Klasický algoritmus pro násobení matic, algoritmus s transpozicí a blokový algoritmus na AMD Epyc 7281 (vlevo) a Apple M1 Pro (vpravo).

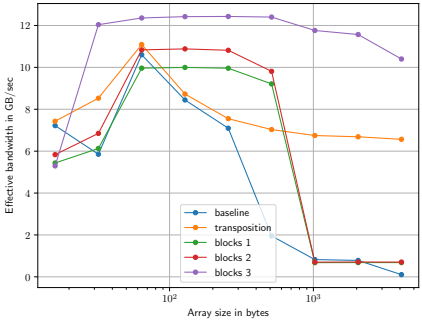
Násobení matic

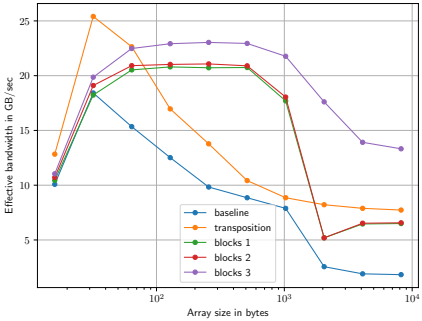
- dál si můžeme všimnout, že prohozením smyček pro `j1` a `k1` lze počet operací dále zredukovat (ukazatel `_m2` spravujeme nešikovně)

Násobení matic

```
template< int blockSize = 16 >
void multiplyMatricesBlockwise3( const ArrayMatrix< Real >& m1,
                                 const ArrayMatrix< Real >& m2 )
{
   memset( this->matrix, 0, size * size * sizeof( Real  ) );
   for( int i = 0; i < size; i += blockSize )
     for( int j = 0; j < size; j += blockSize )
        for( int k = 0; k < size; k += blockSize )
        {
           Real* res = &this->matrix[ i * size + j ];
           Real* _m1 = &m1.matrix[ i*size + k ];
           for( int i1 = 0;
                i1 < blockSize;
                i1++, res += size, _m1 += size )
           {
              Real* _m2 = &m2.matrix[ k*size + j ];
              for( int k1 = 0;
                   k1 < blockSize;
                   k1++, _m2 += size )
                 for( int j1 = 0; j1 < blockSize; j1++ )
                    res[ j1 ] += _m1[ k1 ] * _m2[ j1 ];
           }
        }
}
```

Násobení matic

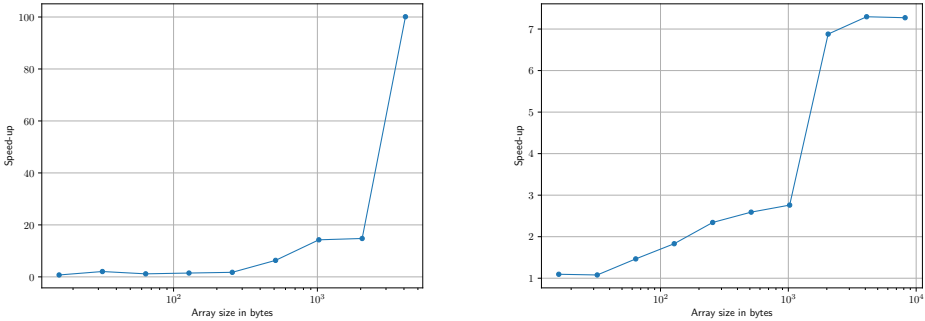




Obrázek: Klasický algoritmus pro násobení matic, algoritmus s transpozicí a blokový algoritmus na AMD Epyc 7281 (vlevo) a Apple M1 Pro (vpravo).

Násobení matic

- tento příklad nám dobře ukazuje, že pokud efektivně využíváme L1 cache, vyplatí se eliminovat i zdánlivé drobnosti
- na této úrovni už se vyplatí znát i assembler a sledovat kód generovaný překladačem



Obrázek: Celkové urychlení optimalizace násobení matic na AMD Epyc 7281 (vlevo) a Apple M1 Pro (vpravo).

Násobení matic

- při operacích s maticemi je efektivní využití cache naprosto zásadní
- mnoho operací je efektivně implementováno v řadě numerických knihoven jako:
  - Blas, Lapack, Atlas, PETSc, Intel MKL
- obecně lze vhodným přeuspořádáním for cyklů výrazně navýšit výkon aplikace
- ukážeme si ještě, jaké výsledky nám dají profilery
- testy byly provedeny na Intel Core i7 4600M
  - 32 kB L1, 256 kB L2, 4MB L3

Násobení matic

Efekt přepínače `-g` překladače `gcc`.

size	baseline		blocks 3	
	with `-g`	without `-g`	with `-g`	without `-g`
16	2.68e-6	2.69e-6	1.50e-6	1.52e-6
32	2.15e-5	2.16e-5	1.21e-5	1.21e-5
64	1.9e-4	1.91e-4	9.62e-5	9.68e-5
128	1.6e-3	1.68e-3	7.75e-4	7.75e-4
256	1.9e-2	1.88e-2	6.20e-3	6.19e-3
512	0.19	0.19	5.46e-2	5.43e-2
1024	3.66	3.40	0.58	0.57

Násobení matic

Efekt přepínače odstranění optimalizací přepínačem `-O3` překladače `gcc`.

size	baseline		blocks 3	
	with `-O3`	without `-O3`	with `-O3`	without `-O3`
16	2.69e-6	1.55e-5	1.50e-6	1.26e-5
32	2.16e-5	1.15e-4	1.21e-5	9.54e-5
64	1.91e-4	9.46e-4	9.62e-5	7.64e-4
128	1.68e-3	8.22e-3	7.75e-4	6.07e-3
256	1.88e-2	7.76e-2	6.20e-3	4.86e-2
512	0.19	0.76	5.46e-2	0.40
1024	3.40	20.43	0.58	3.32

Násobení matic

Naše měření ukazují následující výsledky (100% = běh všech pěti algoritmů):

algorithm	s `-O3`	bez `-O3`
baseline	37%	50%
transposition	11%	9%
blocks 1	21%	19%
blocks 2	23%	11%
blocks 3	6%	9%

Násobení matic

Gprof

- s optimalizacemi `-O3`

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 50.12      0.01     0.01                             TimerRT::start()
 50.12      0.02     0.01                             TimerRT::getTime()
  0.00      0.02     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z16writeT
  0.00      0.02     0.00        1     0.00     0.00  _GLOBAL__sub_I_defaultTim
  0.00      0.02     0.00        1     0.00     0.00  TimerRT::reset()
```

Násobení matic

Gprof

- bez optimalizací `-O3`

```
1  Flat profile:
2
3  Each sample counts as 0.01 seconds.
4    %   cumulative   self              self     total
5   time   seconds   seconds    calls  us/call  us/call  name
6   32.27     21.00    21.00    76163   275.72   275.72  multiplyMatrices( ... )
7   21.62     35.07    14.07    53769   261.73   261.73  multiplyMatricesBlockwise1<16>( ... )
8   16.47     45.79    10.72    88925   120.55   120.55  multiplyMatricesBlockwise2<16>( ... )
9   14.89     55.48     9.69    79484   121.90   121.90  multiplyMatricesWithTransposition( ..
10  14.84     65.14     9.66    94538   102.17   102.17  multiplyMatricesBlockwise3<16>( ... )
11   0.00     65.14     0.00   392984     0.00     0.00  TimerRT::stop()
12   0.00     65.14     0.00   392984     0.00     0.00  TimerRT::start()
13   0.00     65.14     0.00   392949     0.00     0.00  TimerRT::getTime()
```

Násobení matic

Oprofile

- s optimalizacemi `-O3`

```
1  CPU: Intel Haswell microarchitecture, speed 2.901e+06 MHz (estimated)
2  Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask
3  samples  %        image name               symbol name
4  1388122  99.2670  matrix-multiplication    main
5  4297      0.3073  [vdso] (tgid:5205 range:0x7fff393ba000-0x7fff393bbfff) [vdso]
6  1921      0.1374  no-vmlinux               /no-vmlinux
7  1892      0.1353  libc-2.19.so             memset
8  1035      0.0740  matrix-multiplication    TimerRT::getTime()
9  290       0.0207  libc-2.19.so             _int_free
10 258       0.0185  libc-2.19.so             _int_malloc
11 178       0.0127  libstdc++.so.6.0.19      /usr/lib/x86_64-linux-gnu/libstdc++.
12 159       0.0114  libc-2.19.so             __memcmp_sse4_1
13 98        0.0070  libc-2.19.so             malloc
```

Násobení matic

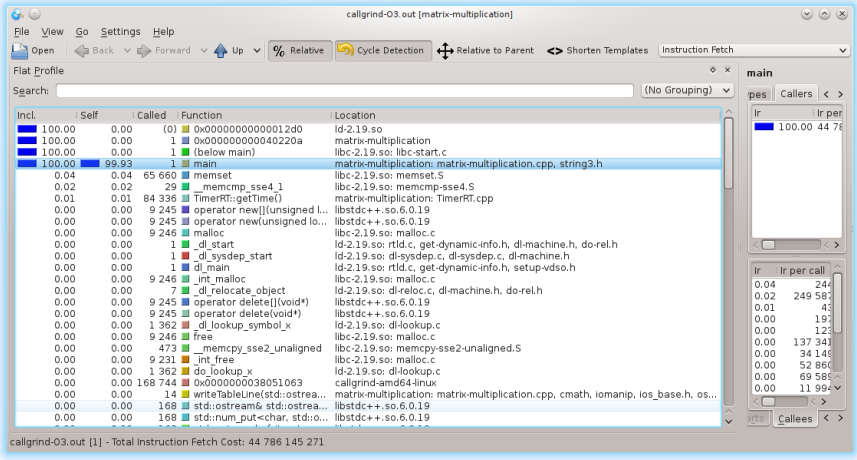
Oprofile

- bez optimalizací `-O3`

```
CPU: Intel Haswell microarchitecture, speed 2.901e+06 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask
samples  %        image name               symbol name
839355   34.9679  matrix-multiplication    multiplyMatrices( ... )
503906   20.9930  matrix-multiplication    multiplyMatricesBlockwise1<16>( ...
367684   15.3179  matrix-multiplication    multiplyMatricesBlockwise2<16>( ...
347259   14.4670  matrix-multiplication    multiplyMatricesWithTransposition( .
337396   14.0561  matrix-multiplication    multiplyMatricesBlockwise3<16>( ...
2684      0.1118  no-vmlinux               /no-vmlinux
806       0.0336  [vdso] (tgid:5481 range:0x7ffff09ec000-0x7ffff09edfff) [vdso]
343       0.0143  libc-2.19.so             memset
215       0.0090  matrix-multiplication    ArrayMatrix<float>::setValue(float c
```

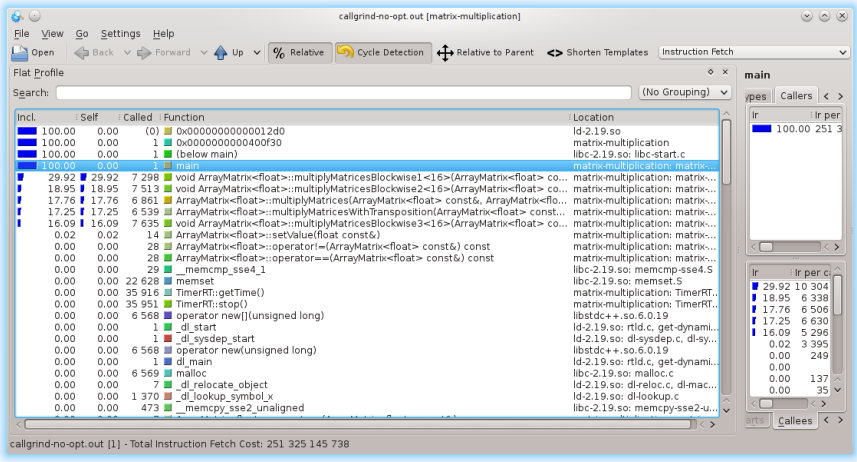
Násobení matic

Callgrind – s optimalizacemi `-O3`



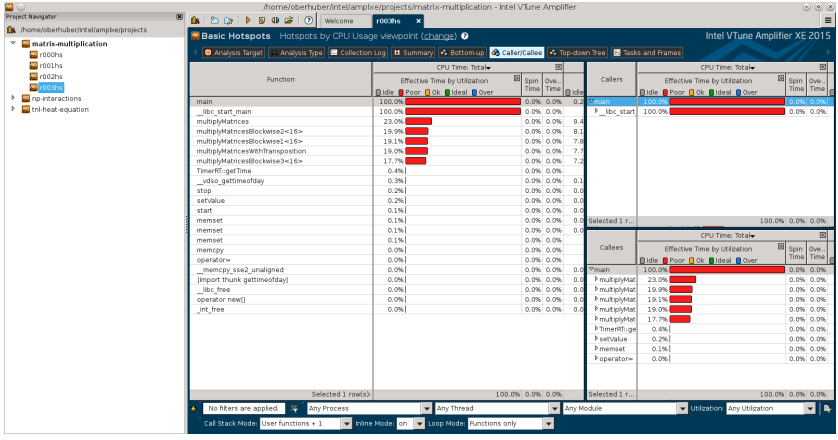
Násobení matic

Callgrind – bez optimalizací `-O3`



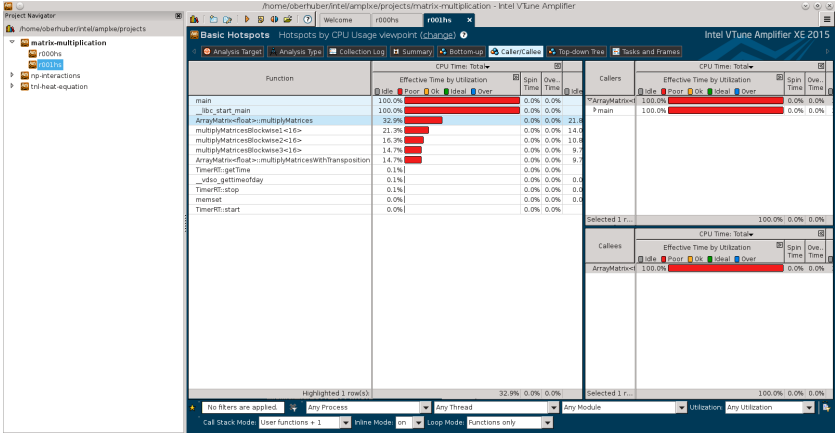
Násobení matic

Intel Vtune Amplifier – s optimalizacema -O3



Násobení matic

Intel Vtune Amplifier – bez optimalizací `-O3`



Násobení matic

Algorithm	Code		Gprof		Oprofile		Callgrind		Vtune	
	-O3	–	-O3	–	-O3	–	-O3	–	-O3	–
Baseline	37%	50%	N/A	32%	N/A	35%	N/A	18%	23%	32%
Transposition	11%	9%	N/A	15%	N/A	14%	N/A	17%	19%	14%
Blocks 1	21%	19%	N/A	22%	N/A	21%	N/A	30%	19%	21%
Blocks 2	23%	11%	N/A	16%	N/A	15%	N/A	19%	20%	16%
Blocks 3	6%	9%	N/A	15%	N/A	14%	N/A	16%	17%	15%

Vidíme, že:

- mnoho profilerů dává zcestné výsledky při zapnutí optimalizací překladače
- bez optimalizací běží výpočet pomaleji a neefektivita využití cache se tím pádem tolik neprojeví
- lepším nástrojem proto může být `cachegrind`
  - `valgrind --tool=cachegrind program args`
  - `kcachegrind`

Transpozice matic

Příklad:
Vraťme se ještě k transpozici matic.

```
1  void transpose()
2  {
3     Real tmp;
4     for( int i = 1; i < size; i++ )
5        for( int j = 0; j < i; j++ )
6        {
7           tmp = this->matrix[ i * size + j ];
8           this->matrix[ i * size + j ] = this->matrix[ j * size + i ];
9           this->matrix[ j * size + i ] = tmp;
10       }
11 }
```

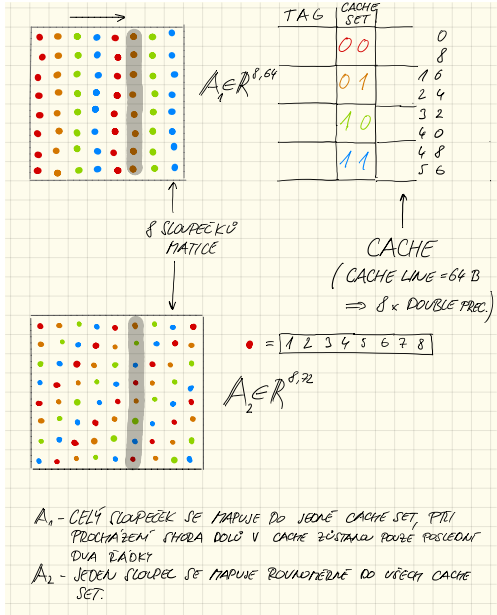
- testování tohoto kódu nám dá následující výsledky
- (měřeno na Apple M1 Pro)

Transpozice matic

Rozměry	Čas	Bandwidth	CPU takty
7	0.000001	0.127134	0
8	0.000001	0.164351	9.049784
9	0.000001	0.206254	7.579637
15	0.000001	0.562406	3.472698
16	0.000001	0.641803	3.173336
17	0.000001	0.719997	2.955381
31	0.000002	2.198364	1.732244
32	0.000002	2.324534	1.67896
33	0.000002	2.361825	1.784656
63	0.000002	6.362348	1.494498
64	0.000002	6.354009	1.560315
65	0.000002	6.481021	1.538898
127	0.000005	11.32839	1.507892
128	0.000007	**9.317155**	**1.972891**
129	0.000006	10.79591	1.578349

Rozměry	Čas	Bandwidth	CPU takty
255	0.000018	13.57795	1.548704
256	0.000115	**2.121634**	**10.83356**
257	0.000018	13.33932	1.615839
511	0.000066	14.80031	1.539382
512	0.000726	**1.345321**	**17.28307**
513	0.000068	14.33781	1.589079
1023	0.000324	12.04092	1.923862
1024	0.003423	**1.141061**	**20.40058**
1025	0.000357	10.95878	2.09418
2047	0.00201	7.764675	2.991541
2048	0.016059	**0.972994**	**23.92793**
2049	0.002796	5.593798	4.15809

Transpozice matic



Transpozice matic

- ukazuje se, že zvětšení rozměrů matice o jedna může zkrátit dobu výpočtu téměř na třetinu
- důvodem je mapování adres do cache
- důležitý je tzv. *critical stride*
  - ten udává, po kolika bytech se budou data mapovat do stejné cache set
- pokud je rozměr matice roven násobku kritického kroku, budou se při procházení jednoho sloupce matice mapovat všechny prvky do omezeného počtu cache set a cache nebude využita efektivně
- řešením by opět bylo provést transpozici po blocích, čímž by se změnilo pořadí, ve kterém k prvkům matice přistupujeme
- tento příklad také ukazuje, že někdy se může vyplatit prokládat prvky polí prázdnými políčky, což zabrání zmíněnému efektu

Dynamická alokace paměti

- jde o velice náročnou operaci, kdy operační systém musí procházet systémové tabulky a najít vhodný volný blok paměti
  - http://www.root.cz/clanky/jak-funguje-malloc-a-free
- u vícevláknových aplikací probíhá alokace sériové a může tak snížit efektivitu paralelizace
- častá dynamická alokace může vést k fragmentaci haldy
- pamětí přitom dnes není nutné šetřit
- může být výhodnější alokovat si veškerou paměť na začátku výpočtu, a pak dynamickou alokaci neprovádět vůbec
- lze také použít funkci `alloca` pro alokaci na zásobníku
  - maximální velikost zásobníku bývá 8kB (`ulimit -a`)

Dynamická alokace paměti

- provedeme test následujících funkcí

```
1  int a[ 512 ];
2
3  void globalArrayTest( int pos )
4  {
5     a[ pos - 1 ] = 1;
6  }
7
8  void localArrayTest( int pos )
9  {
10    int b[ 512 ];
11    b[ pos - 1 ] = 1;
12 }
13
14 void stackTest( int pos )
15 {
16    int* c = (int* )  alloca( pos * sizeof( int ) );
17    c[ pos - 1 ] = 1;
18 }
19
20 void heapTest( int pos )
21 {
22    int* c = (int* ) malloc( pos * sizeof( int ) );
23    c[ pos - 1 ] = 1;
24    free( ( void* ) c );
25 }
```

Dynamická alokace paměti

Výsledek je:

- AMD Ryzen 5 2600 a gcc 12.2

```
1  The global array test took:    4.65 CPU cycles.
2  The local array test took:     6.34 CPU cycles.
3  The stack test took:           6.36 CPU cycles.
4  The heap test took:          105.116 CPU cycles.
```

- AMD Phenom 2 1075T a gcc 4.8

```
1  The global array test took: 14.657 CPU cycles.
2  The local array test took:  13.986 CPU cycles.
3  The stack test took:         50.682 CPU cycles.
4  The heap test took:         178.054 CPU cycles.
```

Shrnutí

Naše pozorování lze shrnout do následujících bodů:

- **data by měla být v paměti uložena tak, aby se k nim přistupovalo sekvenčně a ne náhodně**
- **pokud to tak nejde, zkusíme výpočet rozdělit na menší bloky dat, které se vejdou do cache CPU**
- **data načtená do cache se snažíme využít co nejvíc**