Atak młotowy

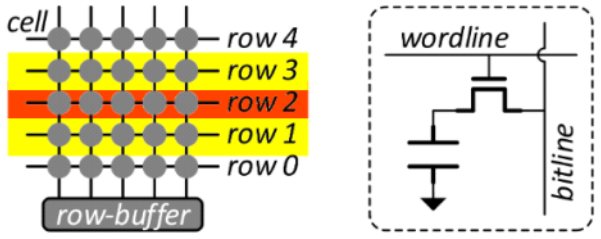
piątek, 5 lipca 2024 11:16

Uwaga! Wykład nie jest zupełnie od początku.
Jeśli chodzi o ten temat lepiej więc zajrzeć do internetu.

Atak „młotowy” (*ang. Rowhammer attack*)

S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," 2003 Symposium on Security and Privacy, 2003., Berkeley, CA, USA, 2003, pp. 154-165, doi: 10.1109/SECPRI.2003.1199334.

Y. Kim et al., "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, 2014, pp. 361-372, doi: 10.1109/ISCA.2014.6853210.



a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

```
1 code1a:
2   mov (X), %eax
3   mov (Y), %ebx
4   clflush (X)
5   clflush (Y)
6   mfence
7   jmp code1a
```

a. Induces errors

```
1 code1b:
2   mov (X), %eax
3   clflush (X)
4
5
6   mfence
7   jmp code1b
```

b. Does not induce errors

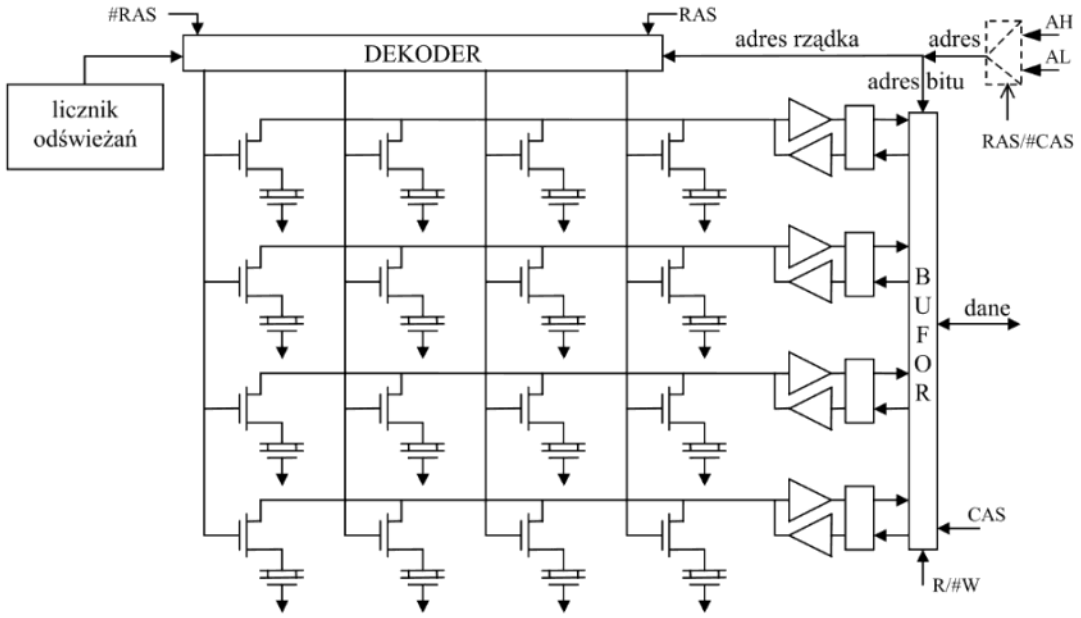
Code 1. Assembly code executed on Intel/AMD machines

20

Wykonując odpowiednio dużo odpowiednio dobranych zmian w wierszach sąsiadujących można spowodować, że wartości w tym wierszu środkowym znacznie szybciej się naładują/rozładują co wynika ze specyfikacji pamięci.
W ten sposób wymyślono taki sposób dość skutecznego ataku na w zasadzie wszystko , bo jeśli można zmieniać dowolne bity w dowolnym miejscu w pamięci to można potem wiedząc mniej więcej co tam jest wymusić cokolwiek.

Ważne jest to, że na przykładzie tego ataku możemy się zastanowić jak się używa takiej pamięci.

Tomczak przypomina ten schemat:



Organizacja pamięci dynamicznej.
RAS – strob adresu rządka, CAS – strob adresu danej

Podczas dostępu do macierzy:

1. jeden wiersz jest aktywowany czyli przepisywany do wewnętrznych buforów
2. Wszystkie operacje po aktywacji są wykonywane na wewnętrznych buforach
3. Po wykonaniu tego wszystkiego dany wiersz jest z powrotem zapisywany do bufora

Jeżeli dany wiersz w danym banku jest aktywny to możemy go odczytać i do niego pisać. Po zakończeniu operacji trzeba go przeładować. Wysłać z powrotem do macierzy.

W ataku dwa wiersze (na żółto) są nie tylko odczytywanie i zapisywane, ale muszą być cały czas aktywowane i deaktywowane, bo gdybyśmy tylko zapisywali to to by się wszystko działo jedynie w buforze. Nie w matrycy. Dlatego oprócz tego, że są wykonywane zapisy do pamięci to opróżniane są bufory wewnątrz procesora (związane z pamięcią podręczną) oraz jest wymuszana taką specjalna instrukcją procesora pełna transakcja na magistrali pamięci. Instrukcja mfence oznacza, że wszystkie wcześniejsze operacje, które były wykonane na pamięci wewnątrz procesora muszą być wysłane do pamięci. Bo normalnie procesor stara się nie operować na pamięci, bo to jest wolne.

Dopiero taka konstrukcja kodu powoduje, że co obieg pętli zmieniany wartości tych dwóch wierszy na żółto), czyli co obieg pętli te dwa wiersze są aktywowane i dezaktywowane.

I dopiero wtedy w środkowym wierszu można spowodować, że jakieś bity zostaną przekłamane.

Pytanie: A dlaczego nie można po prostu zmodyfikować środkowego wiersza?
Odpowiedź: Bo nie mamy do niego dostępu. Właśnie chodzi o to zmieniamy obszar pamięci, do którego nie mamy dostępu.

Pytanie: A jak można na niego wpłynąć, bo ten kod wykonuje właśnie tylko dezaktywowanie tego wiersza pierwszego i trzeciego?

Odpowiedź: Przez zjawiska elektryczne, które zachodzą w otoczeniu, przez indukcję.

Pytanie: Czyli to nie jest precyzyjne?
Odpowiedź: Dość precyzyjne, jeżeli wiemy że w danym miejscu jest zero, a w tych komórkach stale będziemy załadowywali jedynki spowodujemy, że do tego bitu bardziej będzie wpływał ładunek z tych sąsiednich.

No co to jest kondensator?
To są dwa przewodniki oddzielone izolatorem, może zostać zrobiony celowo, albo przypadkiem. Dwa druty puszczone w powietrzu tworzą kondensator. Pomiędzy nimi prąd przepływa.

Pytanie: Czym różni się cflush od mfence
Odpowiedź: cflush dotyczy pamięci podręcznej, a mfence dotyczy wszystkich buforów. O tych dwóch instrukcjach będzie przy okazji pamięci podręcznej i podsystemów w procesorze. Na razie skupiamy się tylko na macierzy czyli budowie pamięci dynamicznej.
……
……
Tomczak powtarza w tym miejscu informacje z poprzedniego wykładu (zaczynając od sekcji Z perspektywy programisty przez obliczenia z prawa Little-a kończąc na układzie FPGA ze strony 39)
……
……

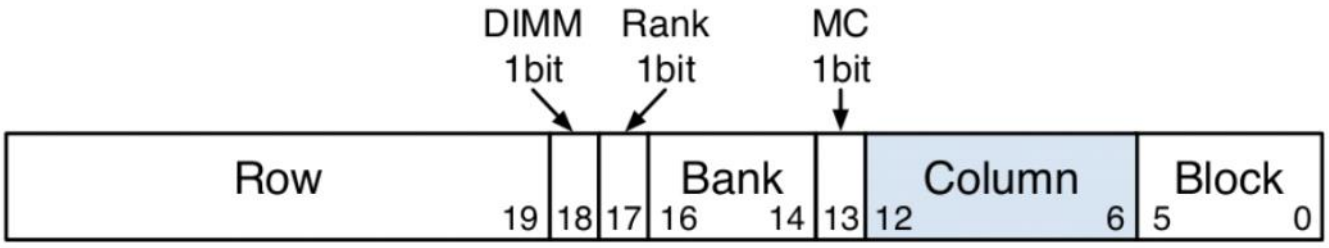
MAPOWANIE ADRESU

piątek, 5 lipca 2024 12:31

Do tej pory nie zajmowaliśmy się w ogóle tym jak nasze adresy używane w programie są tłumaczone na numery kolumny w konkretnym wierszu w konkretnym banku na konkretnym module w konkretnym RZĄDKU.

Mamy do procesora podłączone jeden lub więcej kanałów pamięci. W każdym kanale pamięci może być zamontowany jeden lub więcej modułów pamięci, każdy moduł pamięci może mieć jeden lub więcej RZĄDKÓW - RANK-ów, w każdym RZĄDKU mamy strukturę złożoną z wielu banków pogrupowanych w grupy i w każdym banku jest ileś wierszy i każdy wiersz zawiera ileś kolumn.

I teraz trzeba jakoś zadecydować jak adres wytwarzany (uwaga! Uproszczenie!) przez procesor jest tłumaczony na te poszczególne numery:



Dimitris Kaseridis, Jeffrey Stuecheli, Lizy Kurian John ,Minimalist open-page: a DRAM page-mode scheduling policy for the many-core era. Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). Association for Computing Machinery, New York, NY, USA, 2011, 24–35. DOI:https://doi.org/10.1145/2155620.2155624

Modułu, RZĄDKA na module, kontrolera, kolumny, banku, wiersza w banku i numeru bajtu z danej kolumny.
Są stosowane przeróżne kombinacje!
W związku z atakiem młotowym część nowych rozwiązań stosuje pseudolosowe mapowanie żeby było trudniej zgadnąć, które adresy wystawiane przez procesor tłumaczą się na które numery wierszy, w których bajtach.
Im będzie trudniej ten adres uzyskać tym trudniej będzie przygotować atak, który zmieni konkretne bity w konkretnych komórkach. Jest to jedna z możliwości.

Machine No.	Microarch.	DRAM		Bank Address Functions	Row Bits	Column Bits
		Type, Size	Config.			
No.1	Sandy Bridge i5-2400	DDR3, 8GiB	2, 1, 1, 8	(6), (14, 17), (15, 18), (16, 19)	17∼32	0∼5, 7∼13
No.2	Ivy Bridge i5-3230M	DDR3, 8GiB	2, 1, 2, 8	(14, 18), (15, 19), (16, 20), (17, 21), (7, 8, 9, 12, 13, 18, 19)	18∼32	0∼6, 8∼13
No.3	Ivy Bridge i5-3230M	DDR3, 4GiB	1, 1, 2, 8	(13, 17), (14, 18), (15, 19), (16, 20)	17∼31	0∼12
No.4	Haswell i5-4210U	DDR3, 4GiB	1, 1, 1, 8	(13, 16), (14, 17), (15, 18)	16∼31	0∼12
No.5	Haswell i7-4790	DDR3, 16GiB	2, 1, 2, 8	(14, 18), (15, 19), (16, 20), (17, 21), (7, 8, 9, 12, 13, 18, 19)	18∼32	0∼6, 8∼13
No.6	Skylake i5-6600	DDR4, 16GiB	2, 1, 2, 16	(7, 14), (15, 19), (16, 20), (17, 21), (18, 22), (8, 9, 12, 13, 18, 19)	19∼33	0∼7, 9∼13
No.7	Skylake i5-6200U	DDR4, 4GiB	1, 1, 1, 8	(6, 13), (14, 16), (15, 17)	16∼31	0∼12
No.8	Coffee Lake i5-9400	DDR4, 8GiB	1, 1, 1, 16	(6 13), (14 17), (15 18), (16, 19)	17∼32	0∼12
No.9	Coffe Lake i5-9400	DDR4, 16GiB	2, 1, 2, 16	(7, 14), (15, 19), (16, 20), (17, 21), (18, 22), (8, 9, 12, 13, 18, 19)	19∼33	0∼7, 9∼13

TABLE II: Reverse-Engineered DRAM Mappings on 9 different machine settings. (The **Config.** column presents a specific

No.9	Coffe Lake i5-9400	DDR4, 16GiB	2, 1, 2, 16	(7, 14), (15, 19), (16, 20), (17, 21), (18, 22), (8, 9, 12, 13, 18, 19)	19∼33	0∼7, 9∼13

TABLE II: Reverse-Engineered DRAM Mappings on 9 different machine settings. (The **Config.** column presents a specific DRAM configuration in a quadruple: (channel (#), DIMMs (#) per channel, ranks (#) per DIMM, banks (#) per rank).)

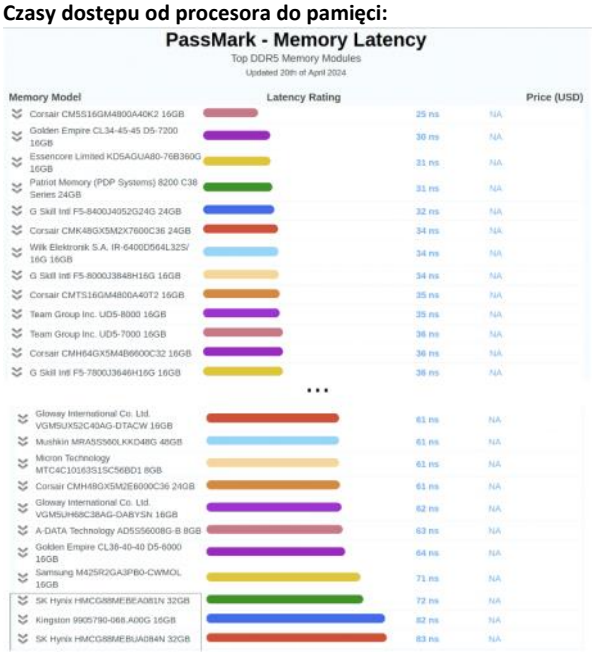
Różne maszyny stosują różne rozwiązania. Tutaj można sobie przejrzeć.
Tu jest odwołanie do artykułu, który warto poczytać, w tym artykule jak Państwo popatrzycie:
. Wang, Z. Zhang, Y. Cheng and S. Nepal, "DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address
Mapping," 2020 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2020, pp. 1-6,
doi: 10.1109/DAC18072.2020.9218599

Ciekawe informacje można wydobyć nawet nie wiedząc do końca co tu się dzieje.
To znaczy:
Numery kolumny to zazwyczaj są bity najmniej znaczące. Czyli jak będziecie Państwo generowali kolejne adresy to mamy dużą szansę, że będziemy trafiali kolejne elementy z wiersza. Ale numery wiersza są daleko. Jak będziemy się poruszać po kolei to następny wiersz nie będzie z tego samego banku tylko z jakiegoś innego. To pozwala przy dostępie sekwencyjnym mieć równocześnie otwartych na raz wiele banków. I wtedy można te koszty otwarcia zamortyzować i one się wtedy nie są od siebie zależne. Z tym, że rzeczywista postać funkcji jest dość trudna, bo producenci stosują różne sztuczki zarówno po to, żeby mieć wydajność wysoką jak i po to, żeby zwiększyć bezpieczeństwo.

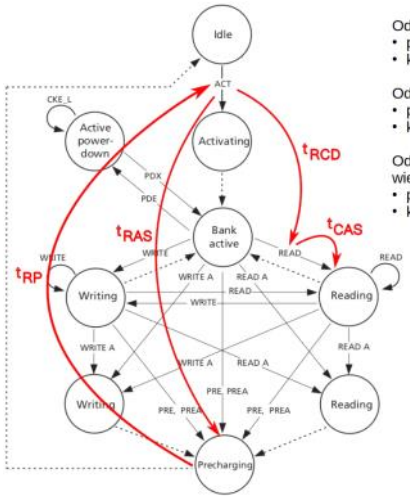
Rzeczywiste parametry pamięci

piątek, 5 lipca 2024 13:03

Czasy dostępu od procesora do pamięci:



To są czasy obejmujące zarówno przejścia po tym automacie:



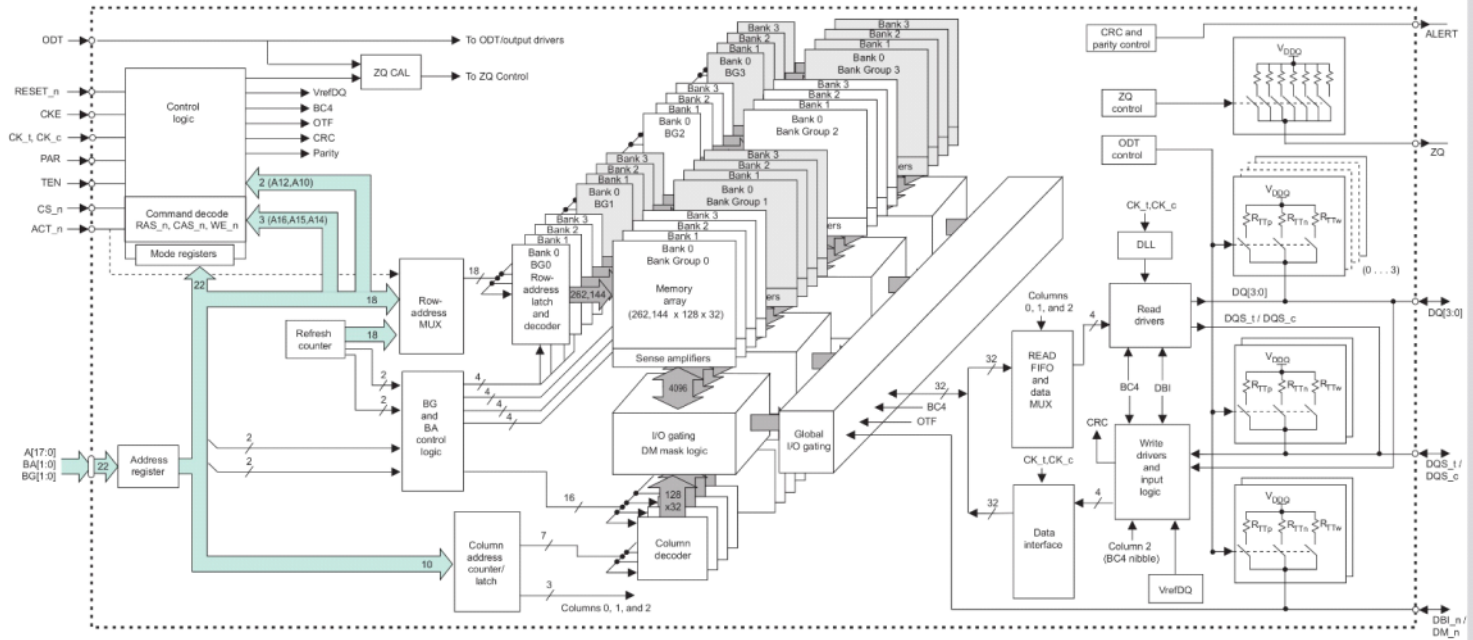
Odczyt danych z aktywnego wiersza:
- pierwszy po t_{CAS}
- kolejne możliwe bez opóźnień

Odczyt danych z nieaktywnego wiersza:
- pierwszy po $t_{RCD} + t_{CAS}$
- kolejne możliwe bez opóźnień

Odczyt danych gdy aktywny jest inny wiersz:
- pierwszy po $t_{RP} + t_{RCD} + t_{CAS}$
- kolejne możliwe bez opóźnień

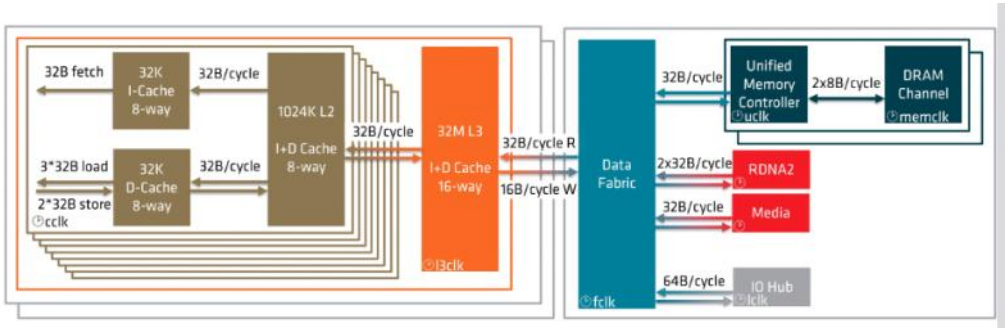
35

Jak i wszystkie narzuty związane z tym co dzieje się w środku:

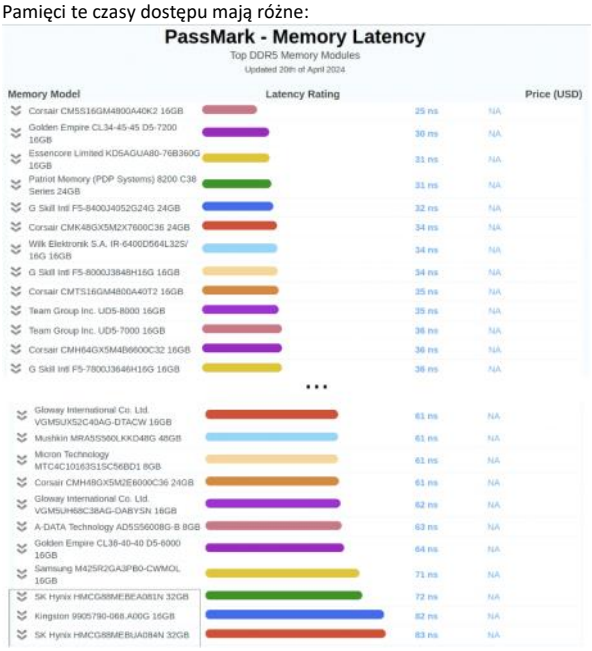


I to są czasy dostępu, które biorą pod uwagę zarówno ten kilkunasto-nano-sekundowy narzut wynikający z tego, że jak wystawimy adres kolumny w

wierszu musimy "pi razy oko 40" cykli magistrali poczekać na rozpoczęcie transferu co daje te kilkanaście-nano-sekund przy najnowszych DDR5 + wewnętrzne czasy wynikające z budowy tego układu łączącego rdzeń procesora z pamięcią:



Pamięci te czasy dostępu mają różne:



Ale na pewno nie w takim zakresie!
Czyli nie od 25-80 ns.
Mogą mieć od kilkunastu do dwudziestu-paru nanosekund czasu dostępu do pamięci.
Cała reszta to jest czas stracony wewnątrz procesora!!!
Tomczak podejrzewa, że to co tu wyświetla to narzędzie to najkrótszy lub uśredniony krótki czas z dostępów przy jakiejś sekwencji odwołań.
Za moment pokaże taki kod.

Pomiary

„Pogoń za wskaźnikiem” (ang. pointer-chasing)

```
volatile S * p = ps ;
auto tBegin = Clock::now() ;
//  **** Pointer chase ****  //
while (true)
{
  p = p->next ;
  if (NULL == p) break ;
}
auto tEnd = Clock::now() ;
```

```
 7 struct S
 8 {
 9   S * next ;
10   uint8_t padding [64 - sizeof (S*)] ;
11 } __attribute__ ((aligned (64))) ;
```

```
131d: e8 fe fd ff ff        call   1120 <std::chrono::_V2::system_clock::now()@plt>
1322: 49 89 c7              mov    %rax,%r15
1325: 4c 89 e8              mov    %r13,%rax
1328: 0f 1f 84 00 00 00 00  nopl   0x0(%rax,%rax,1)
132f: 00
1330: 48 8b 00              mov    (%rax),%rax
1333: 48 85 c0              test   %rax,%rax
1336: 75 f8                 jne    1330 <main+0x130>
1338: e8 e3 fd ff ff        call   1120 <std::chrono::_V2::system_clock::now()@plt>
```

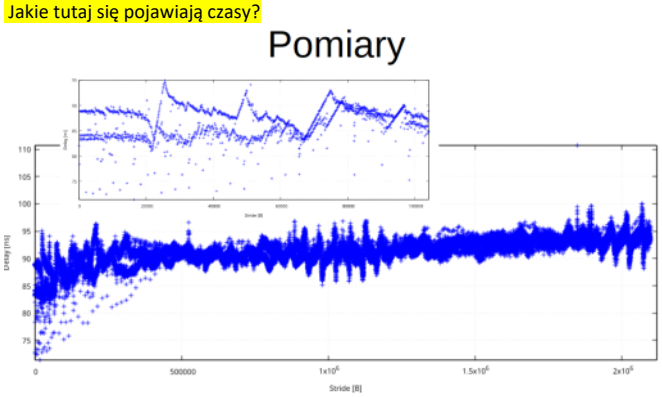
44

Ten kod pozwala takie informacje uzyskać.
Ten kod jest kodem rzeczywistym.
Chcąc zmierzyć taki czas dostępu trzeba zagwarantować, że się kolejne operacje na pamięci nie nałożą. Z tego powodu ten kod jest tak skonstruowany, że odczytuje listę takich struktur. Te struktury zawierają tylko

wskaźnik na następną strukturę oraz trochę bajtów pozwalających na rozepchnięcie rozmiaru struktury do rozmiaru między innymi jednej transakcji na magistrali.
Ta pętla odczytuje kolejne elementy tej listy posługując się wskaźnikami zawartymi w elementach tej listy. Nie da się odczytać kolejnego elementu listy dopóki nie zakończymy odczytu poprzedniego elementu listy. Czas przeczytania fragmentu listy jest sumą czasów dostępów do poszczególnych elementów listy, bo one nie mogą się tutaj na siebie nałożyć.

Kompilator zrobił z tego bardzo ładną pętlę, która po prostu odczytuje zawartość komórki pamięci i tę zawartość traktuje jako adres do następnego odczytu. No i tyle, tutaj w zasadzie się nic nie da przyspieszyć.

Jakie tutaj się pojawiają czasy?



Procesor: Ryzen 9 7945HX 5 GHz
Pamięć: dwa kanały, KF556S40IB-32, 5.2 GT/s, 38-38-38
2 * 5.2 GT/s * 8 B = 83.2 GB/s

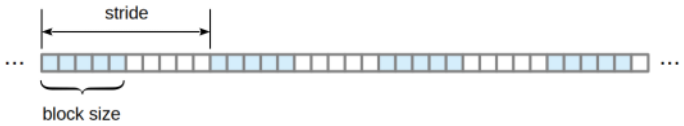
45

```
$ wrmsr -a 0xc0000108 0x2F  # UWAGA!!! Tylko dla Ryzen 9 7945HX !!!
```

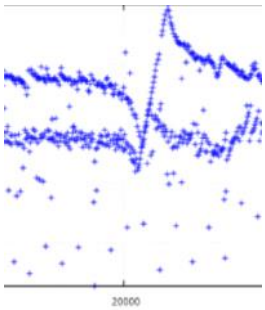
Dla tego procesora i takiej pamięci czasy dostępu są różne natomiast w okolicach 80, 93, 94 ns.

Jak byśmy sobie dokładniej popatrzyli da się jakąś strukturę czasami zauważyć, że w danym miejscu coś dzieje się wolniej więc prawdopodobnie jest tam jakiś konflikt. Nie wiemy gdzie, ale być może w pamięci.

Jak czytać ten wykres?
Na osi pionowej mamy średnie czasy dostępu.
Natomiast na osi poziomej jest odległość między kolejnymi odczytywanymi adresami.



Kiedy ta odległość jest na poziomie około dwudziestu kilobajtów,



czas dostępu jest wyraźnie wyższy niż kiedy ta odległość jest 2 czy 3 kilobajty mniejsza.

Czyli prawdopodobnie tutaj dużo częściej trafiamy na sytuację, że musimy przeładować wiersz z tego samego bajtu. Jakoś tak ta funkcja mieszająca adresy wystawione przez procesor na numer wiersza kolumny itd. Jest skonstruowana, że dla tej długości ma jakieś konflikty.
To jest odległość między kolejnymi adresami.

Ta nasza lista zajmuje obszar kilkadziesiąt gigabajtów pamięci i te elementy są tak porozrzucane.

Te wskaźniki wskazują na elementy odległe o np. 22 kilobajty.
Przy czym to jest oczywiście wielokrotność 64 bajtów.

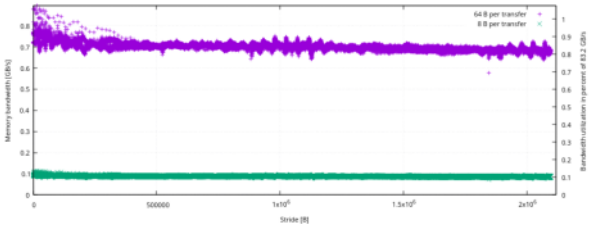
Dla starych systemów - DDR2 bardzo ładnie było widać schodki i te schodki. I one pokazywały, ze to ładnie regularnie coraz częściej lub coraz rzadziej są

otwierane i zamykane kolejne …(niezrozumiałe 00:23:25). Bo tam to wszystko było prostsze.

Natomiast teraz ze względu na te skomplikowane funkcje mieszające adresy dostajemy coś takiego co średnio jest na jakimś poziomie. I rozrzuty są rzędu kilkunastu %, 20%.
Pomijając sam początek gdzie jest nawet może trochę więcej.

Jeżeli mając dany czas zużyty na wykonanie takiej pętli (czas na przeglądnięcie listy) zmierzymy i policzymy ile w tym czasie przesłaliśmy danych:

Pomiary



Jak obliczyć przepustowość?
Rozmiar wskaźnika: 8 B / transfer, ok. **0.1 GB/s (0.1 % z 83.2 GB/s)**
Rozmiar transakcji DDR5: 64 B / transfer, ok. **0.7 GB/s (0.8 % z 83.2 GB/s)**
W rzeczywistości dodatkowe transfery...
46

To wyjdzie nam, że uzyskaliśmy przepustowość na poziomie albo stu albo około siedmiuset MB/s (UWAGA!) z 83 GB/s danych katalogowych. To jest 1/1000 dostępnej przepustowości pamięci.

Prawo Little'a

$$T = \frac{n}{l} = \frac{1 \;\; trans.64\,B}{90 \;\; ns/trans.} = 11.1\,MT/s = 711\,MB/s$$

$$n = T \cdot l = \frac{83.2 \;\; GB/s}{64 \;\; B} \cdot 90\,ns = 117 \;\; trans.$$

Dlaczego tu jest 100 a tu 700?
W obliczeniach przepustowości były wzięte dokładnie te dan, które były przesłane i tylko rozmiar wskaźnika.

Natomiast w tym drugim oszacowaniu było to bardziej rzeczywiste i został wzięty cały rozmiar tej struktury czyli całej transakcji 64B, bo i tak na magistrali pojawiała się cała transakcja 64B.
W rzeczywistości zostało przesłanych mniej więcej tyle danych z taką przepustowością. Natomiast z punktu widzenia programu tylko tyle danych jest użytecznych.

Gdybyśmy liczyli parametry tej konkretnej implementacji to raczej to by było tym naszym miernikiem szybkości przetwarzania.

Pytanie: Strasznie słabo, a dało by się to 0,1 podnieść w takim razie?
Odpowiedź: Przy tym kodzie nie. Tu po prostu z prawa little-a wynika, że jak jedna transakcja trwa 90 ns i transakcje są wykonywane po kolei to w ciągu sekundy jesteśmy w stanie przesłać 711 MB/s.

Pytanie: A jakbyśmy wykonywali bezsensowne działania tylko po to by przyspieszyć…
Odpowiedź: Jak by to były działania bezsensowne to byśmy przesłali więcej danych z większą przepustowością, ale z tej fioletowej kreski. Trzeba inne struktury danych stosować. Tylko wtedy się da. I tu jest o co walczyć. 3 rzędy wielkości. Tu żaden kompilator nic nie zrobi.

To n to natomiast informacja ile musielibyśmy równocześnie wykonywać transakcji żeby uzyskać pełne 83 GB/s. Ponad sto!
Gdybyśmy skonstruowali taką listę gdzie naraz sprawdzamy 100 wskaźników niezależnych od siebie to moglibyśmy się zbliżyć do maksymalnej przepustowości. Nie wiemy jak ta lista musiałaby wyglądać. Musiałaby być

jakaś taka równoległa. Najprościej podzielić nasza listę na 100 niezależnych i każą przetwarzać niezależnie.

JEŚLI PROBLEM JEST JEDNAK Z NATURY SZEREGOWY TO SIĘ NIC NIE ZROBI!
W TEJ CHWILI WYDAJNOŚĆ SIĘ UZYSKUJE TYLKO RÓWNOLEGŁOŚCIĄ!
INNEJ TECHNIKI NIE MA.

Testy Stream

piątek, 5 lipca 2024 14:11

Testy STREAM robi się pi razy 30 lat.
To są testy, w których się bada szybkość przetwarzania dla 4 takich procedur na wektorach:

```
-------------------------------------------------------
                                per iteration:
name     kernel                  bytes  FLOPS     FLOP/B
-------------------------------------------------------
COPY:    a(i) = b(i)               16     0         0
SCALE:   a(i) = q*b(i)             16     1         0.0625
SUM:     a(i) = b(i) + c(i)        24     1         0.0417
TRIAD:   a(i) = b(i) + q*c(i)      24     2         0.0833
-------------------------------------------------------
```

Te działania, które tutaj są wykonywane mogą być wykonywane równocześnie. Dobrze napisany kod będzie przetwarzał dane z maksymalną przepustowością i każdy element tablicy, dla każdego i można przetwarzać niezależnie od pozostałych elementów.

Za to cechą szczególną tych algorytmów jest to ,że one mają bardzo niska intensywność arytmetyczną. I wiele kodów pisanych przez Państwa może być właśnie w tym zakresie. Bo tu już są dosyć skomplikowane obliczenia. Element wektora * jakaś stała + inny element wektora , wytwarzamy jeszcze inny element wektora. Typowe obliczenie, bardzo niska intensywność arytmetyczna.

Jak w tej chwili wyglądają przepustowości uzyskiwane w najlepszych systemach?

Performance Metrics

Analyze Test Configuration: pts/stream-1.3.x - Type: Triad

Stream 2013-01-17

Type: Triad

OpenBenchmarking.org metrics for this test profile configuration based on 3,421 public results since 31 May 2016 with the latest data as of 20 April 2024.

Below is an overview of the generalized performance for components where there is sufficient statistically significant data based upon user-uploaded results. It is important to keep in mind particularly in the Linux/open-source spa
there can be vastly different OS configurations, with this overview intended to offer just general guidance as to the performance expectations.

COMPONENT	PERCENTILE RANK	# COMPATIBLE PUBLIC RESULTS	MB/S (AVERAGE)
16 x 64 GB 4800MT	100th	3	461522 +/- 11087
1 x 480GB DRAM-6400MT	100th	3	328574 +/- 14358
16 x 8192 MB DDR4-2667MT	100th	10	197235 +/- 10440
16 x GB DDR4-2933MT	99th	3	176429 +/- 1217
16 x 16384 MB DDR4-3200MT	98th	5	140562 +/- 12206
16 x 64GB DDR4-2933MT	97th	3	129821 +/- 234
12 x 16 GB DDR4-2133MT	96th	3	122384 +/- 303
12 x 32 GB DDR4-2400MHz Micron 36ASF4G72PZ-2G6D1	95th	5	115692 +/- 4338
12 x 8192 MB DDR4-2666MT	95th	3	115684 +/- 1251
12 x 32 GB DDR4-2400MHz M393A4K40CB2-CTD	95th	3	112073 +/- 4950
12 x 16384 MB DDR4-2666MT	95th	9	108886 +/- 15373
8 x 32 GB DDR4-3200MT	94th	4	102970 +/- 2476
12 x 16384 MB DDR4-2400MHz Hynix HMA82GR7AFR4N-VK	93rd	4	100766 +/- 5606
12 x 16384 MB DDR4-2400MHz M393A2K40BB2-CTD	92nd	3	97207 +/- 4462

https://openbenchmarking.org/test/pts/stream

16 * 4.8 GT/s * 8 = 614.4 GB/s (dual Xeon Platinum 8480)

49

461.522 / 614.4 = 0.751

To 2013-01-1y to nie jest data uzyskania tych danych tylko data wypuszczenia tej wersji kodu.
To są dane sprzed tygodnia dwóch, dość świeże.
Najszybszy system, który był dostępny na tej stronie uzyskuje około 460 GB/s tak pi razy oko
I to odpowiada 75% maksymalnej przepustowości magistrali tego systemu.
Takie wartości będziecie państwo mieli w praktyce.
Jak trochę lepszy system z mniejszą liczbą procesorów, mniejszą liczbą kanałów pamięci to można przekroczyć 80%, nawet niektórzy dochodzą do 90%. Ale to bardzo mocno zależy od systemu i pamięci.
I tyle się da dla kodów równoległych.

Szybka lista

piątek, 5 lipca 2024 15:21

Jak pisać kod, żeby efektywnie wykorzystywać pamięć?
Przede wszystkim wykonywać jak najmniej transakcji na magistrali.
3 przykłady konstrukcji listy, które charakteryzują się coraz mniejszą liczbą transakcji na magistrali:

```
 1 struct Slow
 2 {
 3   Slow      * next ;
 4   LargeData   data ;
 5 } ;
 6
 7
 8 struct Faster
 9 {
10   Faster    * next ;
11   LargeData * data ;
12 } ;
13
14
15 struct EvenFaster
16 {
17    unsigned    next [MAX_LEN] ;
18    LargeData * data [MAX_LEN] ;
19 } ;
```

- Równoległe transakcje z pamięcią:
  - równoległość na poziomie instrukcji (*ang. instruction-level parallelism*)
  - programowanie równoległe (*ang. parallel programming*)
  - bezpośredni dostęp do pamięci (*ang. direct memory access, DMA*)
- Programowe i sprzętowe pobieranie uprzedzające (*ang. data prefetching*)
- Wzorce dostępu do pamięci (*ang. memory access pattern*) zwiększające lokalność przestrzenną odwołań do pamięci (*ang. spatial locality of references*)
  - kafelkowanie (*ang. tiling*)
  - krzywe wypełniające (*ang. space-filling curves*) krzywa Hilberta, Mortona (krzywa Z)

50

Pierwszy przypadek:

```
struct Slow
{
  Slow      * next ;
  LargeData   data ;
} ;
```

Mamy w każdym elemencie listy wskaźnik na następny element + dużo danych. Każdy element tablicy to osobna transakcja na magistrali. * Wskaźniki są oddalone o rozmiar tego elementu. Taka jedna struktura z definicji języka

zajmuje jeden blok pamięci.

Drugi przypadek:

```
struct Faster
{
  Faster    * next ;
  LargeData * data ;
} ;
```

Tutaj w elementach tablicy przechowujemy wskaźniki na następny element oraz wskaźniki na dane, wtedy rozmiar elementu w liście nam się zmniejsza. Może się zmniejszyć wyraźnie. Tutaj jest ten rozmiar tylko dwa razy większy od elementu.

Trzeci przypadek:

```
struct EvenFaster
{
   unsigned    next [MAX_LEN] ;
   LargeData * data [MAX_LEN] ;
} ;
```

Zamiast listy stosujemy tablicę wskaźników.
TO NAWET NIE SĄ WSKAŹNIKI TYLKO INDEKSY! XD
Przeglądamy indeksy po kolei, a elementy tablicy zawierają indeksy następnego elementu tablicy.
A w drugiej tablicy w zupełnie innym obszarze pamięci znajdują się wskaźniki na dane związane z tymi indeksami. Wtedy przeglądanie listy ogranicza się wyłącznie na operacji na tej jednej tablicy, w której nie ma nic więcej oprócz indeksu następnego elementu tablicy.
I to są takie podstawowe proste sztuczki.

Należy profilować kod. Jeżeli się okaże, ze problemem w kodzie są opóźnienia na pamięci trzeba zacząć szukać struktur danych, w których tych opóźnień będzie mniej. Albo algorytmów, w których będzie mniej opóźnień. Zależy.

Techniki fachowców

piątek, 5 lipca 2024 16:04

- Równoległe transakcje z pamięcią:
  - równoległość na poziomie instrukcji (*ang. instruction-level parallelism*)
  - programowanie równoległe (*ang. parallel programming*)
  - bezpośredni dostęp do pamięci (*ang. direct memory access, DMA*)

Równoległość można realizować na różnych poziomach.

1 POZIOM RÓWNOLEGŁOŚCI!
Równoległość na poziomie instrukcji - to jest mniej więcej poziom np. w tych testach. Czyli jeśli widzieliście instrukcje przetwarzania wektorowego, które niezależnie przetwarzają wiele elementów naraz to wtedy jeśli takich instrukcji będziemy używali to możemy na raz przesyłać więcej danych, które są przetwarzane niezależnie i wzajemnie się nie blokują.

To jest pierwszy poziom równoległości, ale nie jedyny!!!!

2 POZIOM RÓWNOLEGŁOŚCI (w klasycznym sensie)
Pisanie programów, które uruchamiają wiele procesów albo wątków. I wtedy każdy wątek przetwarza swoje zbiory danych niezależnie od pozostałych. Czyli zwiększamy ruch na magistrali.

3 POZIOM RÓWNOLEGŁOŚCI
Używanie specjalnych sprzętowych układów, które wykonują operacje na pamięci w trakcie kiedy robimy inne rzeczy. Te układy nazywamy np. układami DMA. Te układy się uruchamiają i przesyłają dane, a my w tym czasie możemy robić inne rzeczy.

- Programowe i sprzętowe pobieranie uprzedzające (*ang. data prefetching*)

POBIERANIE UPRZEDZAJĄCE (ang. PREFETCHING)
Jak się nie da zrównoleglić tak napisać algorytm żeby mniej więcej było wiadomo jakie dane będą potrzebne za moment. I wtedy pobiera się te dane nawet nie wiedząc czy będą potrzebne. Wyróżniamy dwie techniki:
- Programowa
- Sprzętowa

Tu jest właśnie użycie takiego specjalnego zestawu narzędzi, które wyłączają to pobieranie uprzedzające, gdyby Tomczak go nie wyłączył to te parametry byłyby znacznie lepsze. Chodzi o test pogoni ze wskaźnikiem.

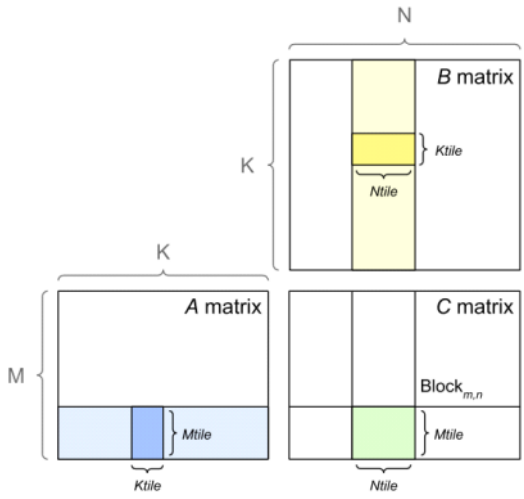
Pytanie: Czyli gdyśmy zamiast tej struktury pogoni za wskaźnikiem użyli tablicy indeksów procesor mógłby sobie zgadnąć sprzętowo, że jeśli robimy bardzo dużą pętlę, w której po kolei odczytujemy te elementy to po prostu zaciągnąć sobie większy chunk tej tabeli?
Odpowiedź: Będzie na kolejnych slajdach, pokażę na konkretnym przykładzie, nie da się tak jednym pytaniem. Ale generalnie to jest dobra technika, która daje efekty czyli

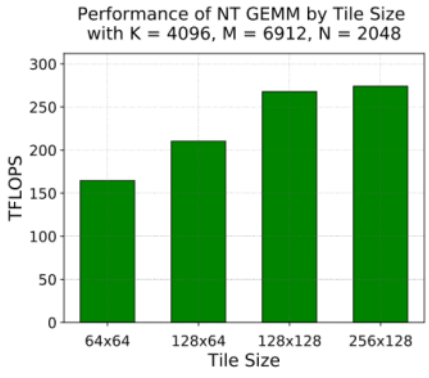
WŁAŚCIWE UŁOŻENIE DANYCH W PAMIĘCI
Jak umieścimy dane często używane w takich wierszach i bankach żeby można je było jednocześnie przesłać to będziemy mieli mniejszy czas dostępu. Tylko trzeba je tak poukładać, żeby tworzyć jak najmniej transakcji wolnych, takich, które wymagają w najgorszym przypadku zamknięcia aktywnego wiersza, aktywowania nowego i dopiero rozpoczęcia na nim jakichś operacji.

Generalnie te wszystkie techniki właściwego układania danych polegają na tym, że dane wykorzystywane blisko siebie w programie powinny być ułożone blisko siebie w pamięci. I to jest technika, która może nie daje efektów najlepszych, ale całkiem dobre, bo jak Państwo widzieliście jak zwiększamy adresy kolejnych elementów w pamięci, kolejnych bajtów to najpierw te konkretne bajty są umieszczane w tym samym wierszu, a potem wiersze są rozrzucane po różnych… (39:40) Czyli tam nie będzie potrzeby otwierania
i zamykania wielokrotnego banku. Jak wyglądają te dwie główne klasy specjalnych sposobów układania danych do pamięci?

KAFELKOWANIE

Jest bardzo często stosowane, sztuczka opisywana w każdym podręczniku do wydajnego programowania.





Kafelkowanie polega na tym, że przy mnożeniu macierzy przetwarzane są NIE kolejne elementy z kolumny macierzy tylko z macierzy wycina się takie prostokątne fragmenty i jest duża szansa, że będą zajmowały one spójny obszar pamięci. I wykonuje się obliczenia dla takich prostokątnych

fragmentów pamięci. Kafelków.

Zamiast zapisywać macierz w postaci takiego ciągu liczb możemy ją podzielić na mniejsze macierze i te macierze zapisać osobno.

ALE TO JEST TYLKO JEDNA WERSJA. JEST WIĘCEJ!
TA KONKRETNA JEST ZWIĄZANA WYŁĄCZNIE Z UŁOŻENIEM DANYCH W PAMIĘCI.

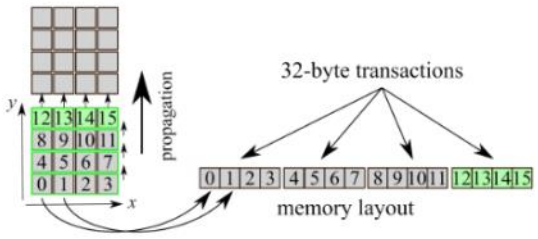


Fig. 5. Propagation in north (N) direction for XYZ memory layout and double precision f_i values (8 bytes per value). Numbers denote linear memory offset of f_i value for corresponding node (for example, f_i for node at $x = 0, y = 1$ is placed at offset 4). Four consecutive double precision f_i values (one row) form a single 32-byte transaction. During propagation, only four 32-byte memory transactions (reads) are required: one from the neighbour tile (read of nodes 12 to 15) and three from the current tile (read of nodes 0 to 11).

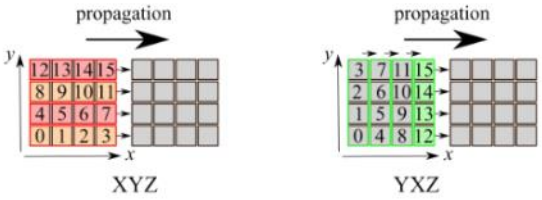


Fig. 6. Propagation in east (E) direction for two memory layouts and double precision f_i values. For XYZ layout, four 32-byte wide memory transactions (marked with red rectangles) are needed to update nodes from the neighbour tile (nodes with indices 3,7,11,15). YXZ layout results in only one 32-byte transaction for nodes from the neighbour tile because now these nodes have close indices (12,13,14,15). Additionally, during the propagation inside the tile, only three (instead of four) 32-byte read transactions are done for nodes with indices 0 to 11. The fourth transaction is done from the neighbour tile.

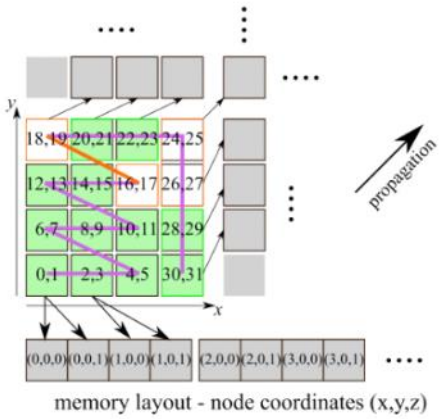


Fig. 7. Propagation in north-east (NE) direction for zigzagNE memory layout and double precision f_i values. In this memory layout, the two consecutive memory locations store f_i values for nodes with the same x and y coordinates — only z coordinate differs. Thus, each square on the picture of tile denotes two 8-byte f_i values placed in neighbour memory locations. Partially utilised (uncoalesced) memory transactions (offsets 16 to 19 and 24 to 27) are marked orange. Each orange transaction is done twice, and only half of the data is used. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

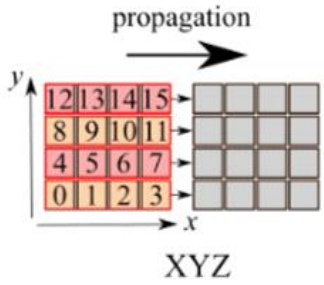
52

Źródło: https://doi.org/10.1016/j.cpc.2018.04.031

Na obrazkach wyżej widać sposób ułożenia danych w pamięci. Liczby to indeksy kolejnych elementów w pamięci dla kafelków 4 x 4 elementy. I dla kafelków 4^3 elementów, takich plasterków w jednym kafelku sześciennym były 4. To nie są macierze tylko przestrzenne struktury danych opisujące geometrię.

Ten program służył do symulacji przepływów cieczy w jakimś środowisku trójwymiarowym
i ta domena obliczeniowa, przez którą płynie ciecz była podzielona na takie małe sześcianiki równomiernie rozłożone. I płyn przepływał pomiędzy tymi sześcianikami.

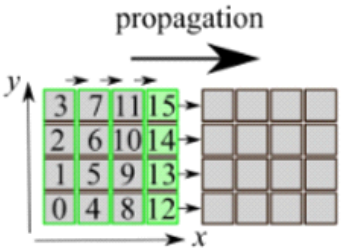
I teraz zamiast te wszystkie sześcianki rozmieścić w pamięci jak się uda to były wycinane z tej całej przestrzeni w takie małe podzbiory 4x4 na 4 najbliższe sąsiednie komórki i dane z tych wycinków były układane w pamięci np. w taki sposób:



XYZ

Wierszami.

lub



YXZ

Kolumnami

W zależności od tego jaki był wzorzec dostępu - czy dane pomiędzy kafelkami były przesyłane w pionie czy w poziomie czy na ukos.

Ciekawostka:
Sposób ułożenia danych, który powoduje, że te skrajne wartości w kafelkach, które są przesyłane do sąsiednich kafelków są ułożone blisko siebie w pamięci.

Jak kogoś interesuje sposób obliczania tego numeru to w artykule jest dostęp:
https://doi.org/10.1016/j.cpc.2018.04.031
Ale generalnie była to stosunkowo prosta funkcja, którą paroma dodawaniami i jakimiś operacjami logicznymi dało się łatwo wytworzyć.

I to jest sposób układania danych w pamięci, który powoduje, że jak się odwołujemy tylko do tych skrajnych danych z kafelka to one będą w pamięci ułożone blisko siebie.
Czyli najprawdopodobniej w jednym otwartym wierszu.

KRZYWE WYPEŁNIAJĄCE

Drugi sposób układania danych w pamięci. Nie wiąże się z kafelkowaniem tylko po prostu indeksowaniem elementów.

Najczęściej stosowanym rozwiązaniem tego typu jest

KRZYWA Z:

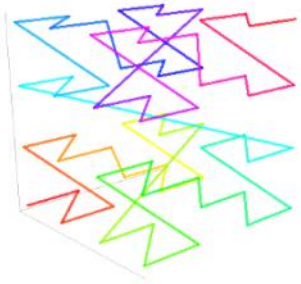
+	0	1	4	5	16	17	20	21
0	0	1	4	5	16	17	20	21
2	2	3	6	7	18	19	22	23
8	8	9	12	13	24	25	28	29
10	10	11	14	15	26	27	30	31
32	32	33	36	37	48	49	52	53
34	34	35	38	39	50	51	54	55
40	40	41	44	45	56	57	60	61
42	42	43	46	47	58	59	62	63

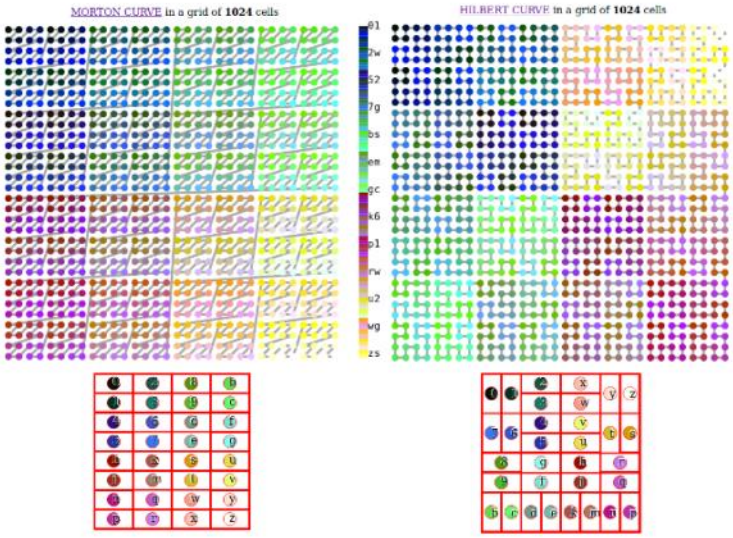
Te dane są ułożone tak, że im są dalsze tym są dalej w pamięci, a im są w bliższym otoczeniu tym są bliżej w pamięci. Krzywa może być też oczywiście trójwymiarowa.
Ta krzywa jest używana najczęściej, bo jest łatwa do implementacji.

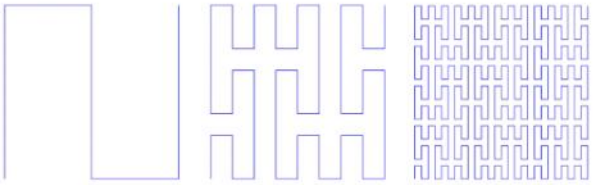
Są też inne krzywe:

Krzywe wypełniające

+	0	1	4	5	16	17	20	21
0	0	1	4	5	16	17	20	21
2	2	3	6	7	18	19	22	23
8	8	9	12	13	24	25	28	29
10	10	11	14	15	26	27	30	31
32	32	33	36	37	48	49	52	53
34	34	35	38	39	50	51	54	55
40	40	41	44	45	56	57	60	61
42	42	43	46	47	58	59	62	63







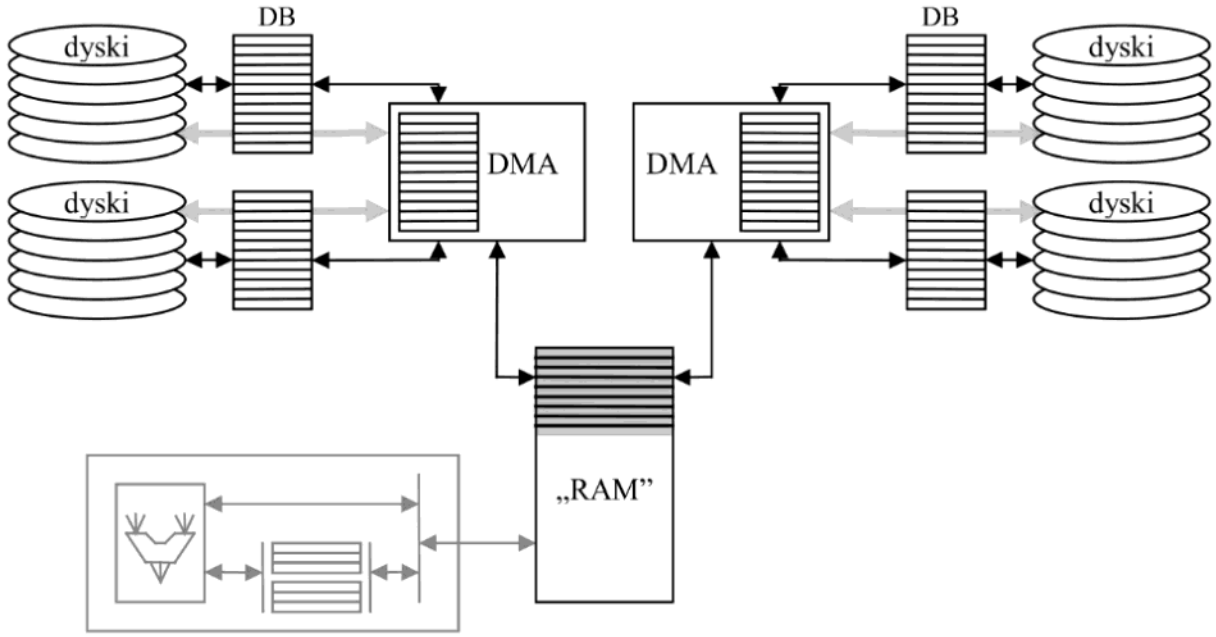
Źródło: https://en.wikipedia.org/wiki/Z-order_curve
https://en.wikipedia.org/wiki/Space-filling_curve

53

DMA

piątek, 5 lipca 2024 17:44

Jak jest zbudowane DMA?



Jest podłączone do pamięci niezależnie od procesora.
Czyli modyfikuje pamięć niezależnie od procesora!
To jest ważne, bo procesor może nie wiedzieć, ze pamięć została zmodyfikowana.
Służy zazwyczaj do przesyłania danych do jakichś innych urządzeń.
Są też układy DMA, które przesyłają dane z pamięci do innej pamięci, ale najczęściej jest ono wykorzystywane do podłączenia jakiegoś zewnętrznego urządzenia do pamięci.

Jak się używa DMA?

1. Najpierw układ sterujący transferem się odpowiednio ustawia:
   - skąd przesłać
   - dokąd przesłać
   - ile przesłać
   - w jaki sposób przesłać
   - co zrobić jak już się prześle czyli jak poinformować procesor, że się przesłało
2. Potem kanał DMA się uruchamia.
3. I czeka się na zakończenie operacji, a w trakcie transferu można robić inne rzeczy.

Jeszcze raz: konfiguracja, uruchamia się i od tego momentu zaczynają się transfery, a jak one trwają to można robić inne rzeczy.

Ważne jest to, że DMA może pracować w różnych trybach! Różnie wpływa na działanie pozostałych elementów podłączonych do pamięci.
Na obrazku wyżej widzimy trzy strzałki do pamięci, natomiast jak było widać wcześniej, moduł pamięci miał tylko jedną magistralę adresową i jedną magistralę danych. Nie mógł na jednej magistrali w tym samym czasie przesyłać danych z kilku źródeł. Chyba, że zastosujemy tryby pracy.

Tryby DMA:

1. Blokowy (tryb domyślny w którym uruchamiamy układ DMA) on przesyła dane z maksymalną przepustowością, a procesor w tym czasie może przejść w tryb niskiego poboru energii
2. Podkradanie cykli - Powodujący, ze nasz sterownik DMA przesyła dane przez krótki czas, a potem zwalnia magistralę dla innych urządzeń. Wtedy transfer jest z mniejszą wydajnością, ale system nie staje.
3. Przezroczysty - można tak zaprogramować sterowniki DMA, że będą przesyłały dane kiedy magistrala nie jest używana.

To chyba szczerze lepiej tłumaczy:

- Tryby pracy DMA
  - blokowy (*ang. burst mode*) – pełna blokada magistrali pamięci podczas transferu DMA
  - podkradanie cykli (*ang. cycle stealing mode*) – częste blokowanie magistrali na krótki czas
  - przezroczysty (*ang. transparent/hidden mode*) – transfer DMA tylko gdy magistrala pamięci nie jest używana przez CPU

55

Układ DMA jest tak skonstruowany, że jest w stanie przesyłać dane z bardzo wysoką przepustowością.

Pytanie: Kiedy inne urządzenia korzystają z DMA i magistrala pamięci jest zablokowana dla procesora to procesor wykonuje sobie inne operacje, które nie wymagają dostępu do pamięci, a kiedy magistrala zostaje zwolniona to on wysyła te dane za jednym razem?
Odpowiedź: Nie, jak jest magistrala zablokowana i procesor napotyka na instrukcję, która wymaga dostępu do pamięci to się zatrzymuje i czeka aż się magistrala zwolni. Ale zazwyczaj DMA się używa tak, że jeżeli planujemy transfer blokowy to wtedy uruchamiamy DMA, a procesor przełączamy w tryb niskiego poboru energii czyli go zatrzymujemy. I on wtedy nie robi nic, żeby nie pobierać energii. Chyba, że mamy fragment kodu, który np. liczy tylko na rejestrach. Ale i tak musi ten kod mieć. Więc gdyby to była króciutka pętla, która liczy tylko na rejestrach i jest wewnątrz procesora zmagazynowana to

wtedy może się wykonywać.

Przyszłość pamięci DDR

piątek, 5 lipca 2024 21:14

JEDEC DDR Generations				
	DDR5	DDR4	DDR3	LPDDR5
Max Die Density	64 Gbit	16 Gbit	4 Gbit	32 Gbit
Max UDIMM Size (DSDR)	128 GB	32 GB	8 GB	N/A
Max Data Rate	6.4 Gbps	3.2 Gbps	1.6 Gbps	6.4Gbps
Channels	2	1	1	1
Total Width (Non-ECC)	64-bits (2x32-bit)	64-bits	64-bits	16-bits
Banks (Per Group)	4	4	8	16
Bank Groups	8/4	4/2	1	4
Burst Length	BL16	BL8	BL8	BL16
Voltage (Vdd)	1.1v	1.2v	1.5v	1.05v
Vddq	1.1v	1.2v	1.5v	0.5v

W tej chwili jesteśmy w DDR5.
Maksymalny rozmiar modułów to w tej chwili wydaje się 128GB. To jest 4 razy więcej niż poprzednim razem.
Maksymalna przepustowość to się mówi w tej chwili o 8800Gb/s. Czyli to by trzeba odpowiednio przeskalować.
Ważne jest to, że nowe pamięci DDR5 mają dwa kanały.

Jeśli popatrzymy sobie na moduł DDR4 albo wcześniejsze to one mają 64 lub 72-bitową szynę danych (w zależności czy z korekcją czy bez) i szynę adresową i są widziane jako jeden moduł pamięci.
Natomiast moduły DDR5 maja dwa 32-bitowe kanały, bo tam są kody korekcyjne. Te dwa kanały są niezależne. Także na jednej płytce pamięci, na jednym module pamięci już mamy dwa kanały.

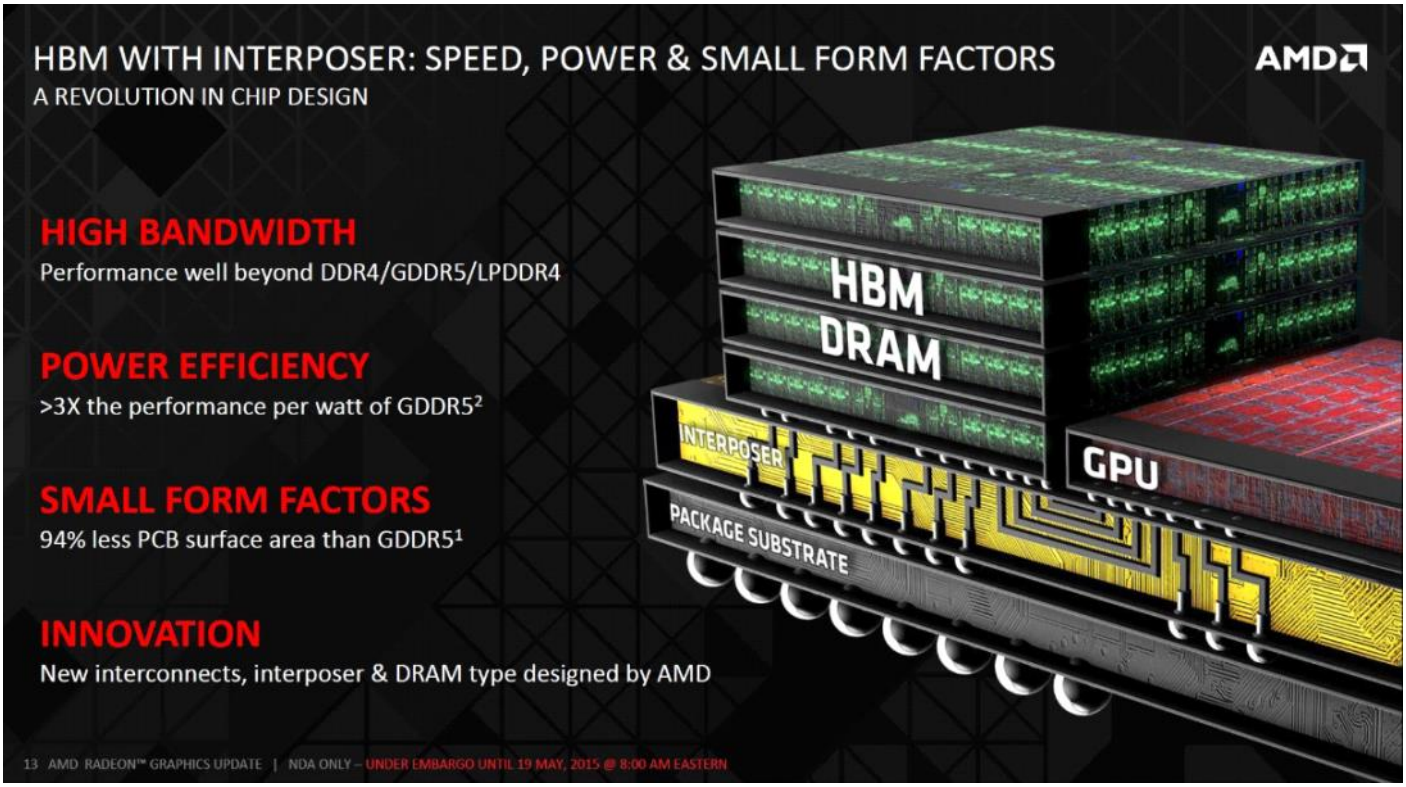
DDR5 ma już tak szybką magistralę, że nie wystarczyło by danych z jednego kanału, żeby go wysycić w ośmiu transakcjach.

W pamięciach DDR5 transakcje trwają dwa razy dłużej niż w pamięciach DDR4.
W DDR4 - 8
W DDR5 - 16.
Bo magistrala jest dwa razy szybsza.
A wszystkie opóźnienia w pamięci są takie jakie były.
Im wyższa przepustowość tym musi być więcej transakcji na magistrali albo czas jeden transakcji musi być niższy.

W zastosowaniach szybkich stosowane są różne warianty pamięci dynamicznych np. w kartach graficznych wolnych w tej chwili są stosowane pamięci graficzne dynamiczne.
Z naszego punktu widzenia może być interesujące to, że tam są zazwyczaj 4 transfery na cykl zegara.
To są pamięci QD quadra - poczwórne. Pamięci graficzne są nastawione na przepustowość.

W szybszych systemach, nie tylko graficznych, stosuje się już pamięci HBM. Już od iluś lat.

Jak one są zbudowane?



Generalnie to są zwykłe pamięci dynamiczne tyle, że płytki krzemowe są sklejone ze sobą w taką kanapkę, żeby połączenia były krótsze i ponieważ, że są krótsze to mogą być szersze i może być ich więcej, w tej chwili szerokość szyny danych takiej pamięci HBM to idzie w Kb. Z tym, że to nie jest jeden kanał tylko kilkanaście kanałów 32b czy tam, w tej chwili szerokości jednego kanału są chyba do 128b.

Czyli pamięci HBM to są zwykłe pamięci DDR tylko, że w wielokanałowe i w jednej obudowie i z krótkimi połączeniami dlatego to może działać szybciej. Tomczak upraszcza, ale dużo nie kłamie.

W najnowszych Xeonach, które mają wyjść w 3 kwartale tego roku mają się pojawić pamięci MCR. Kombinowane ramki na module pamięci, działa to w ten sposób, że dwa RZĄDKI mają po 64 bity na magistrali danych, zapis jest na obu tych RZĄDKACH naraz. I potem na magistrali pamięci są przesyłane z dwa razy większą częstotliwością niż były odczytywane wewnątrz pamięci. W ten sposób pamięci są nadal tak samo wolne jak były, ale magistrala pomiędzy modułęm pamięci a procesorem jest dwa razy szybsza. Już w tej chwili się mówi o 8400 chyba. To jest 2*4200. I już. To jest ten pomysł, który w najnowszych Xeonach ma się pojawić. On da jeszcze więcej przepustowości pamięci od 8800 w górę.
Natomiast na pewno tanie nie będzie!

W tej chwili wszędzie jest równoległość, na każdym kroku, to nie działa dlatego szybciej, bo jest szybciej, ale dlatego, że więcej może przetworzyć na raz. Już od dawna jesteśmy na prawie Gustafsona a nie Amdahla.

JEŚLI CHODZI O PAMIĘCI OPERACYJNE TO W ZASADZIE TYLE!