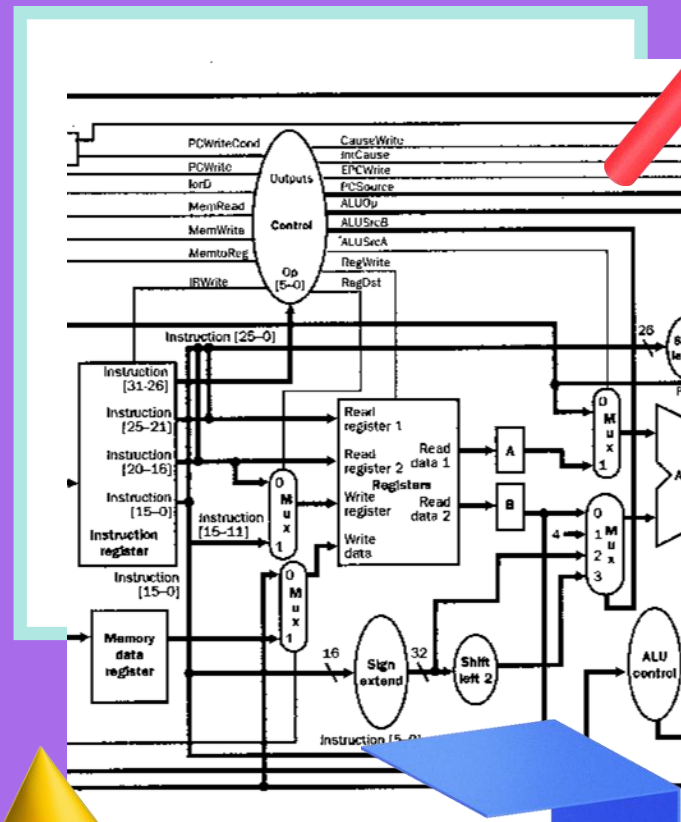


Curs 1 - Optimizări low-level





Conținut

01. Cache

Introducere & terminologie,
Înmulțirea matricelor,
Cache Lines,
Blocking

03. Unrolling Loops & Dependency Chains

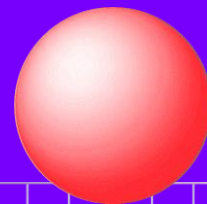
Loop Unrolling
Dependency Chains
SIMD

02. Array-uri locale statice

Array-uri de float-uri,
Array-uri de int-uri,
String-uri

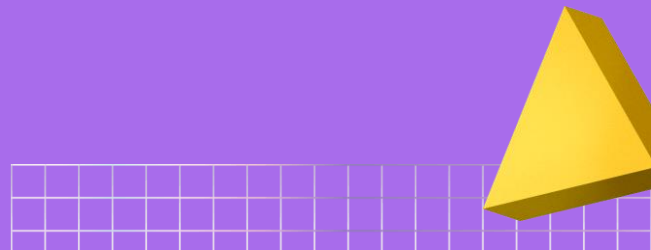
04. Branchless Programming

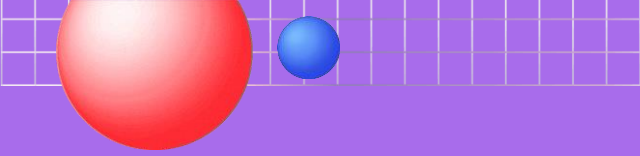
Branch-uri,
Branch Prediction,
Exemple,
Branchless Programming
folosind SIMD





Cache

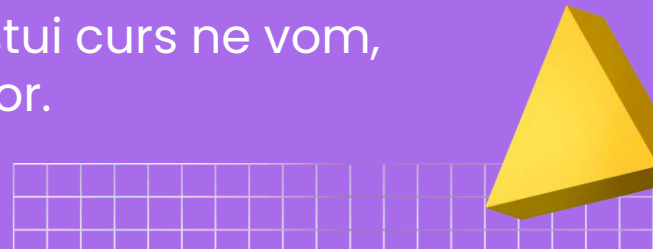


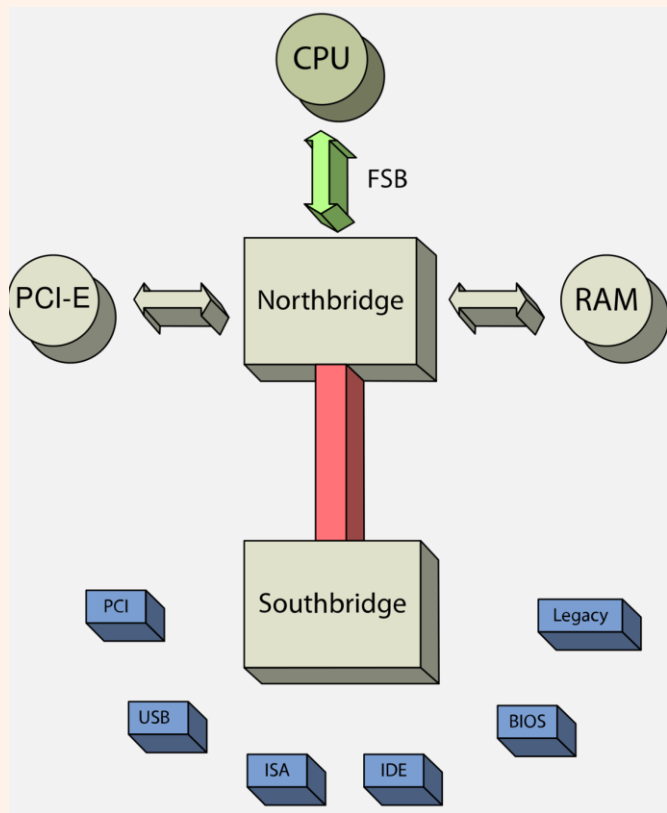


Prin cache ne referim la o zonă de memorie (reducă ca dimensiune) aflată pe procesor, folosită pentru a accesa date folosite frecvent și a minimiza citirile/ scrierile din memoria RAM.

Atunci când solicităm date din memoria RAM, procesorul mai întâi verifică dacă acele date se află în memoria cache. Dacă se află, atunci se operează direct pe datele din memoria cache. Altfel, datele vor fi citite din memoria RAM. Atunci când datele ajung către procesor, acesta le va salva în cache pentru a le putea folosi atunci când acestea vor mai fi necesare.

Există mai multe tipuri de cache. În cadrul acestui curs ne vom referi strict la memoria cache aflată pe procesor.





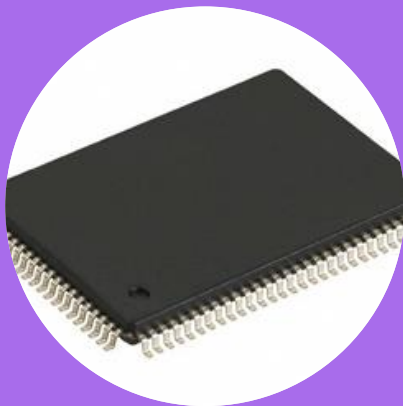
Northbridge

Pentru a prelua date din memoria RAM, acestea trebuie să treacă prin Northbridge pentru a ajunge la procesor. Acest transfer durează mult.

Datele transferate între procesor și placa video trec tot prin Northbridge. Transferul este lent.

Southbridge

Transferul de date între procesor și periferice se realizează prin Southbridge. Acest transfer este mai lent decât cel prin Northbridge.



SRAM

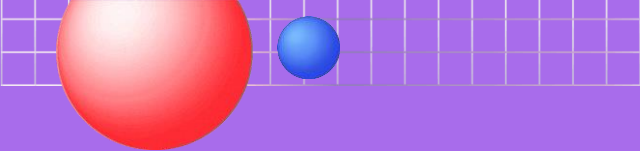
memorie rapidă,
dar scumpă



DRAM

memorie ieftină,
dar lentă

**Memoria cache este de tip SRAM,
iar memoria sistemului (RAM) este de tip DRAM**



Terminologie

Cache Hit:

Datele solicitate se află în cache, deci nu trebuie aduse din memoria RAM. Ne dorim ca programele noastre să aibă cache hit-uri!

Cache Miss:

Datele solicitate nu se află în cache, deci trebuie citite din memoria RAM. Ne dorim să minimizăm numărul de cache miss-uri ale programelor noastre.

Data Cache:

Tip de cache folosit pentru datele cu care lucrează programul.

Instruction Cache:

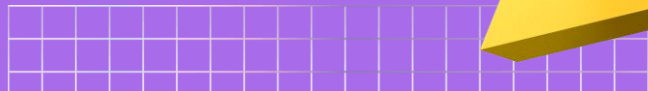
Tip de cache pentru instrucțiunile programului pe care procesorul urmează să le execute.

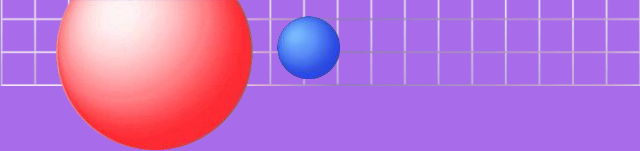
Unified Cache:

Poate fi folosit atât pentru date cât și pentru instrucțiuni.

Ciclu:

Durata unui tick de pe procesor. Este cea mai mică unitate de timp pentru procesor. Un procesor de 1GHz executa un miliard de tick-uri într-o secundă. Latența este numărul de cicluri ale procesorului care se petrec atunci când procesorul execută o acțiune sau așteaptă un rezultat.





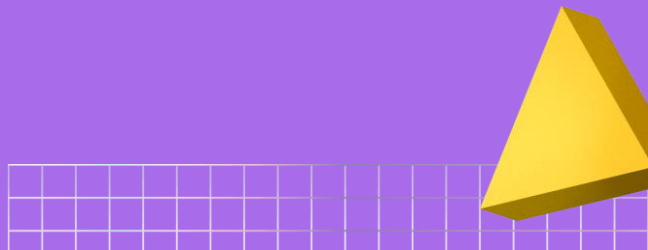
L1 – Cel mai mic cache, dar și cel mai rapid

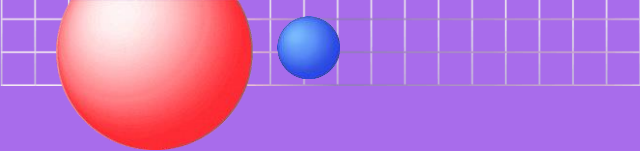
Multe procesore au și Data Cache și Instruction Cache pe acest nivel
Fiecare nucleu are propriul cache la acest nivel

L2 – Dimensiune medie, viteză medie

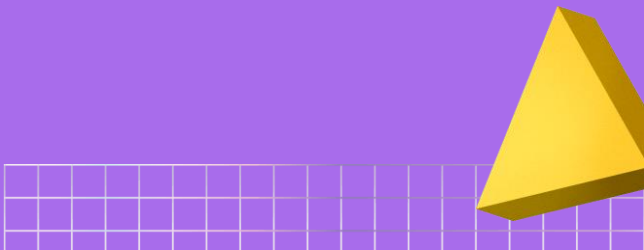
L3 – Dimensiune mare, cel mai lent nivel de cache


Unele procesoare nu au acest nivel de cache
Acest nivel este împărțit de toate nucleele procesorului





Tip de cache	Intel Core i9-13900K	AMD Ryzen 9 7950X	Intel Core i9-12900KF	AMD Ryzen 7 7700X	AMD Ryzen 5 7600X
L1	2.1MB	1MB	1.4MB	512KB	384KB
L2	32MB	16MB	14MB	8MB	6MB
L3	36MB	64MB	30MB	32MB	32MB
Preț	3370 RON	3130 RON	2700 RON	1900 RON	1350 RON

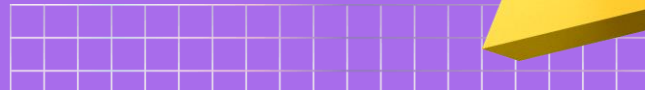




În tabelul de mai se poate vedea timpul de așteptare al procesorului pentru a obține date din diferite tipuri de memorie. Numerele din tabel sunt estimative, dar sunt foarte asemănătoare cu cele reale.

Tip de memorie	Latență (nr. Cicluri)
Registru	1
Cache L1	3
Cache L2	15
Cache L3	60
Memoria sistemului (RAM)	150
Hard Disk	Fără număr

Pentru a accesa orice altă memorie, pentru care este implicată vreo magistrală (PCI, USB..), latența este mult mai mare decât cea pentru nivelele de cache.



Înmulțirea matricelor

Inițializare matrice

```
constexpr long long size = 1000;

const auto a = new double[size * size];
const auto b = new double[size * size];
const auto c = new double[size * size];
// a, b and c are freed at a later point, the code was not included here.

for (auto row = 0; row < size; row++)
{
    for (auto col = 0; col < size; col++)
    {
        const auto a_value = static_cast<double>(rand()) /
                                static_cast<double>(RAND_MAX);
        const auto b_value = static_cast<double>(rand()) /
                                static_cast<double>(RAND_MAX);

        a[row * size + col] = a_value;
        b[row * size + col] = b_value;
    }
}
```

Înmulțire

[illegible]

Înmulțirea matricelor v2

Inițializare matrice

```
constexpr long long size = 1000;

const auto a          = new double[size * size];
const auto b          = new double[size * size];
const auto b_transposed = new double[size * size];
const auto c          = new double[size * size];

for (auto row = 0; row < size; row++)
{
    for (auto col = 0; col < size; col++)
    {
        const auto a_value = static_cast<double>(rand()) /
                               static_cast<double>(RAND_MAX);
        const auto b_value = static_cast<double>(rand()) /
                               static_cast<double>(RAND_MAX);

        a[row * size + col]          = a_value;
        b[row * size + col]          = b_value;
        b_transposed[col * size + row] = b_value;
    }
}
```

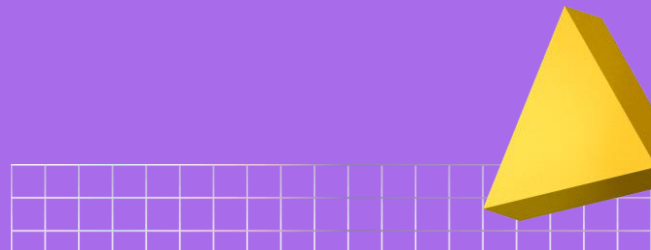
Înmulțire

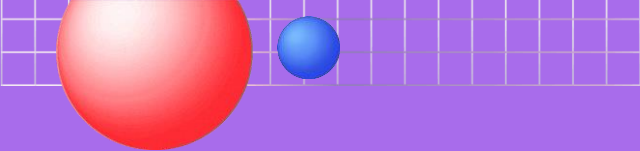
```
for (auto row = 0; row < size; row++)
{
    for (auto col = 0; col < size; col++)
    {
        c[row * size + col] = 0.0;
        for (auto dot = 0; dot < size; dot++)
            c[row * size + col] +=
                a[row * size + dot] *
                b_transposed[col * size + dot];
    }
}
```



**Care variantă este
mai rapidă?**

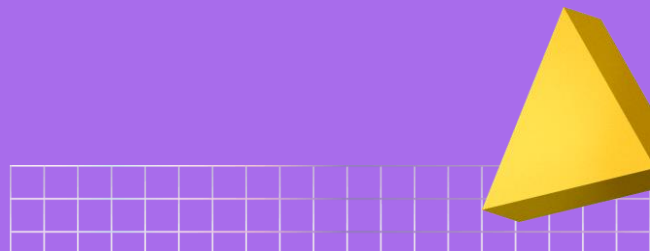
De ce?

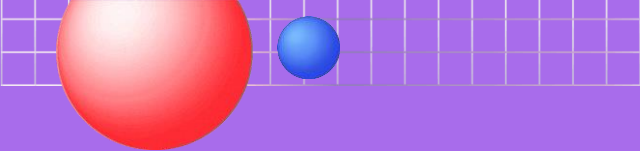




**A doua variantă este mai
rapidă!**

**De aproape 6 ori mai
rapidă!**

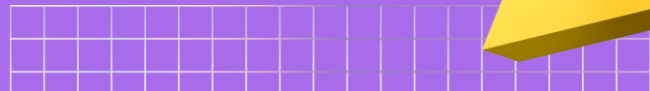




Cache lines

Atunci când procesorul solicită date din memoria RAM, acesta primește datele în bucăți de câte 64 de bytes (sau alte dimensiuni în funcție de procesor). Aceste bucăți se numesc **Cache Lines**. Procesorul nu salvează în cache doar datele solicitate într-un moment, ci un bloc întreg de date adiacente cu care va lucra.

Dacă se solicită date de la o adresă adiacentă de memorie, acestea se află deja în cache și nu mai este nevoie să fie preluate din RAM. Scrierea se efectuează tot în **Cache Line**. Datele vor ajunge în RAM în momentul în care **Cache Line**-ul este șters pentru a fi înlocuit cu alt **Cache Line**.



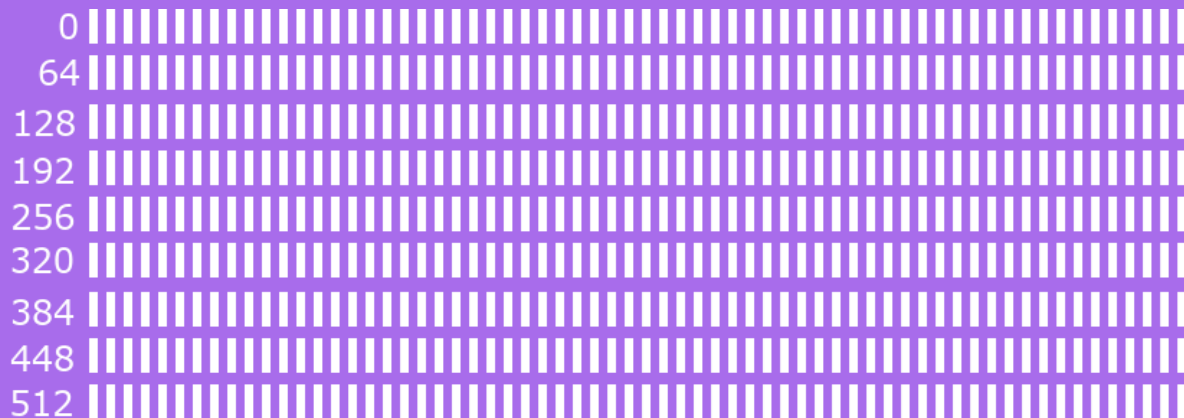


Cache lines

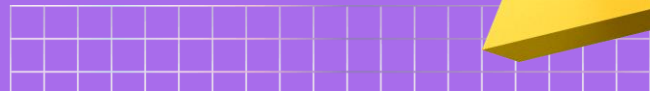
Prima linie

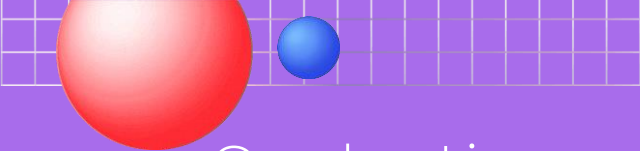
RAM

A doua linie



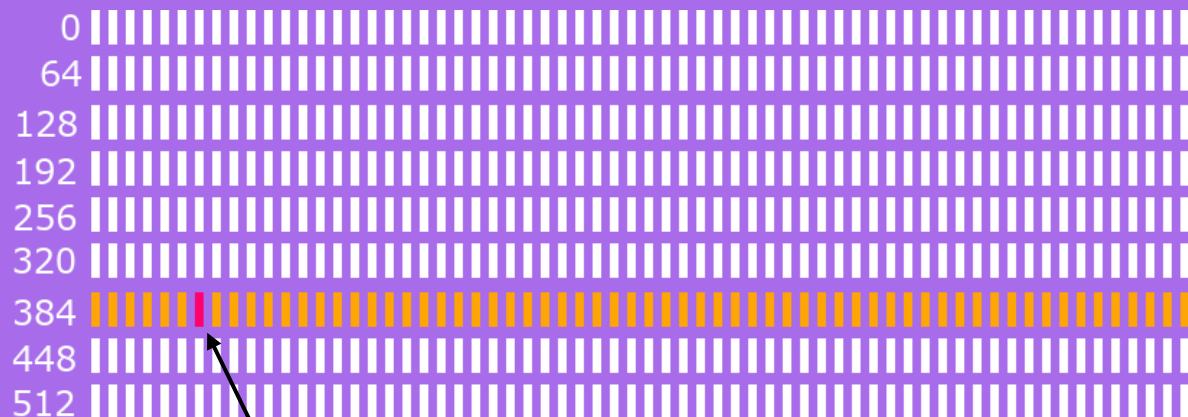
Octeții din RAM sunt ținuti în grupe de câte 64. Prima linie începe la byte-ul 0 și ajunge până la byte-ul 63. Următoarea linie începe de la byte-ul 64 și continuă până la byte-ul 127, etc..





Cache Lines

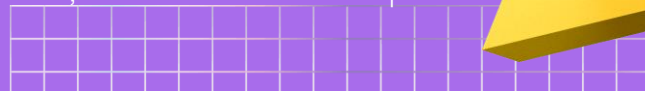
RAM

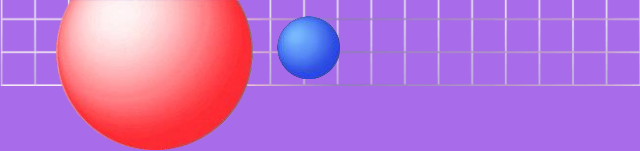


Pentru a determina cache line-ul, adresa byte-ului este împărțită la 64, iar restul este ignorat.

Byte-ul de la adresa $0x187$ (391_{10}) se află pe cache line-ul 6 (pornind de la 0).

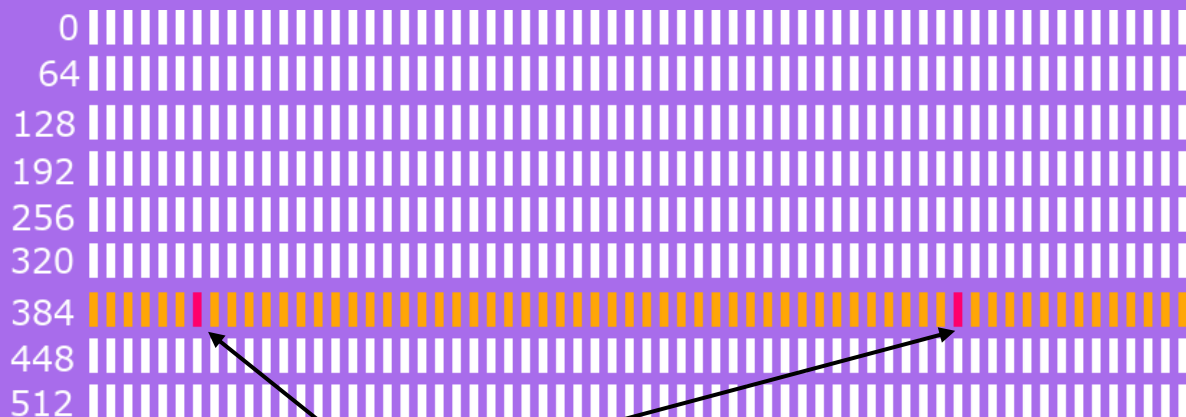
Toți octeții ai căror adresă împărțită la 64 rezultă în același număr se află pe același cache line.



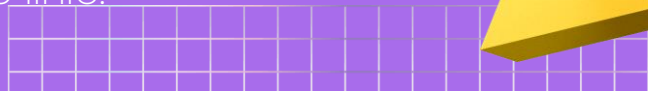


Cache lines

RAM



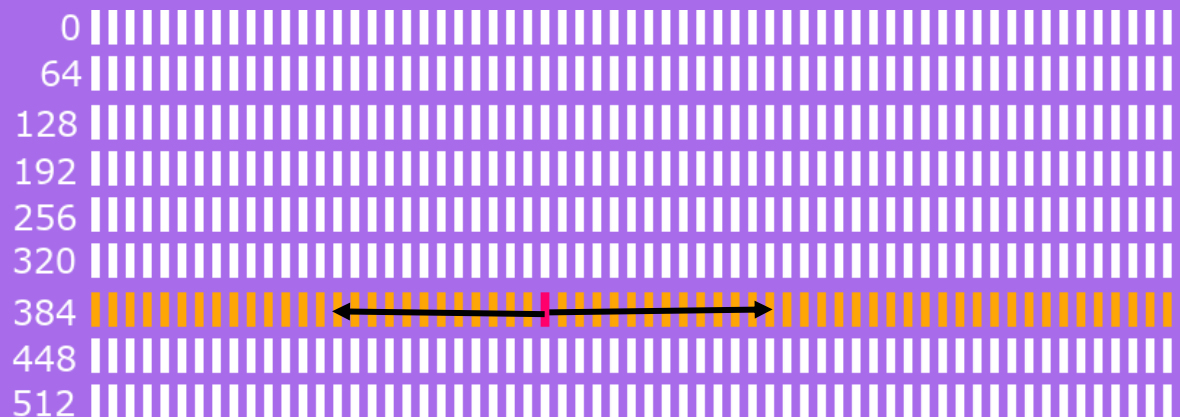
Dacă solicităm succesiv 2 bytes de pe aceeași linie, vom avea inițial un **Cache Miss** pentru primul byte, dar vom avea un **Cache Hit** pentru următorul byte solicitat. Cât timp linia se află în cache-ul procesorului, vom avea **Cache Hit**-uri pentru toți octeții solicitați de pe linie.



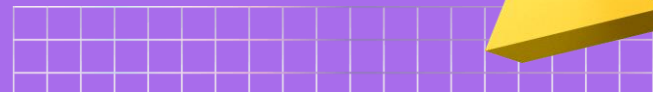


Cache lines

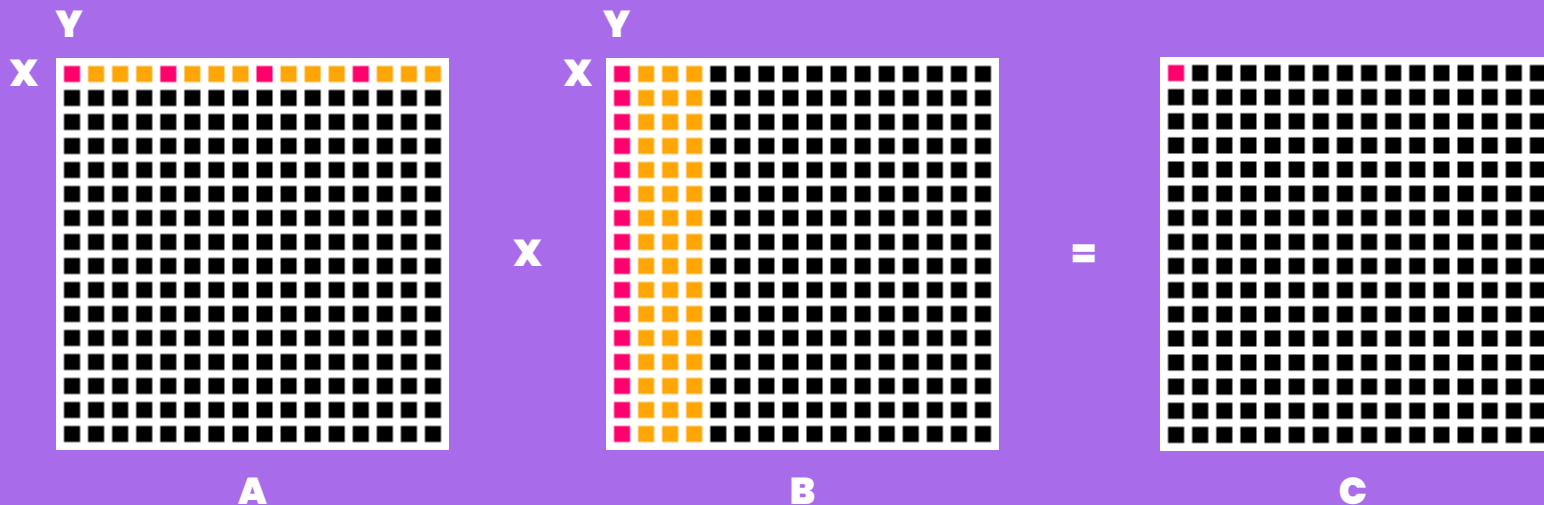
RAM



Atunci când iterăm secvențial o zonă de memorie contiguă, direcția de deplasare este irelevantă deoarece întregul cache line se află în memoria cache.



Cache lines & Înmulțirea matricelor

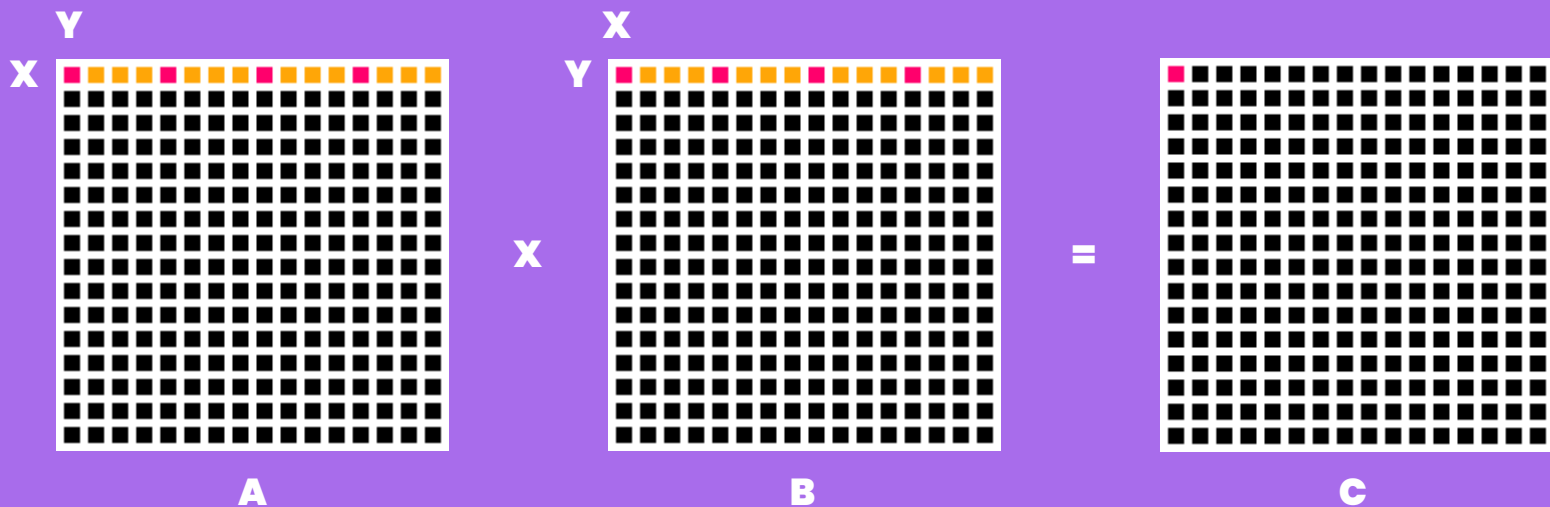


Pentru a calcula un element al matricei rezultante, se ia o linie a primei matrice și o coloană a celeilalte matrice și se realizează produsul scalar al acestora.

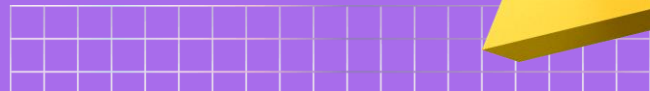
Dacă ambele matrice sunt indexate pe linie, algoritmul nu este eficient la citirea coloanelor din a doua matrice. Dacă ambele matrice sunt indexate pe coloană, accesarea liniilor primei matrice va fi ineficientă.

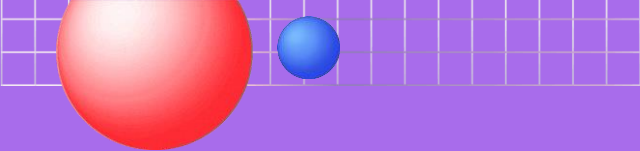


Cache Lines & Înmulțirea matricelor v2



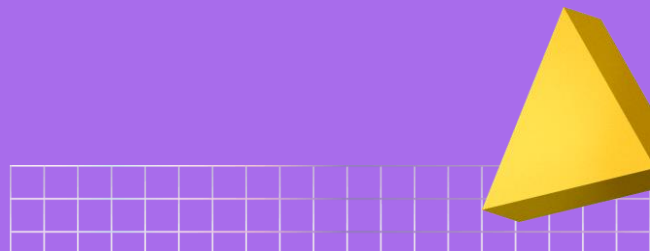
Dacă una dintre matrice este transpusă, atunci putem înmulți eficient cele două matrice, yey!





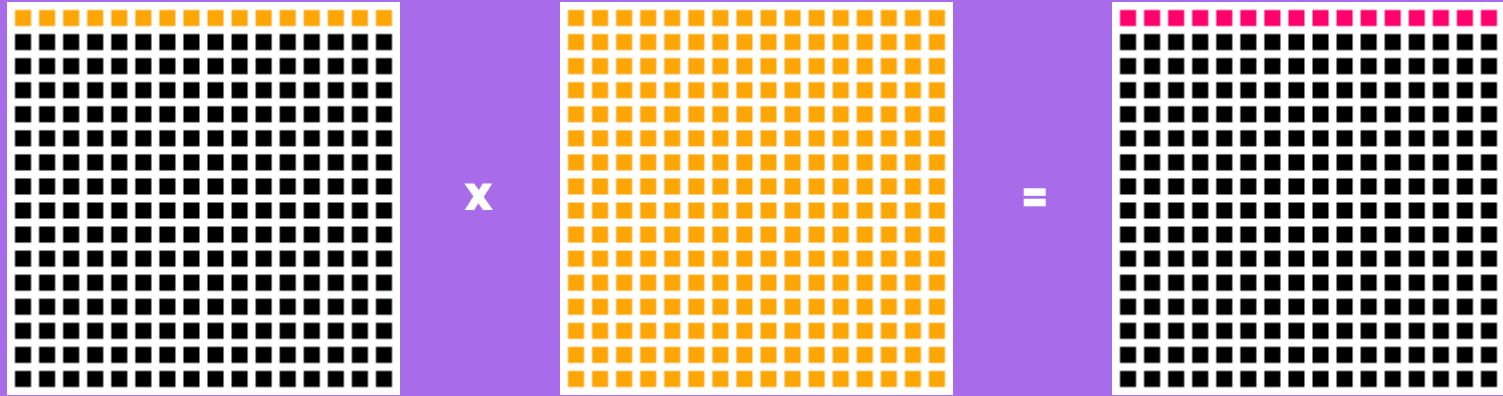
Blocking

Este o tehnică prin care împărțim o problemă în subprobleme mai mici pentru a reduce dimensiunea datelor cu care se lucrează. Astfel, sunt șanse mai mari ca memoria folosită să se afle în cache, iar programul să fie mai eficient.

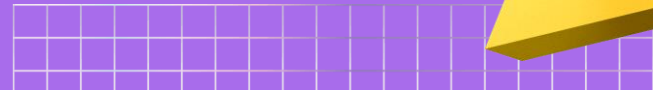




Cache Thrashing/ Pollution & Înmulțirea matricelor

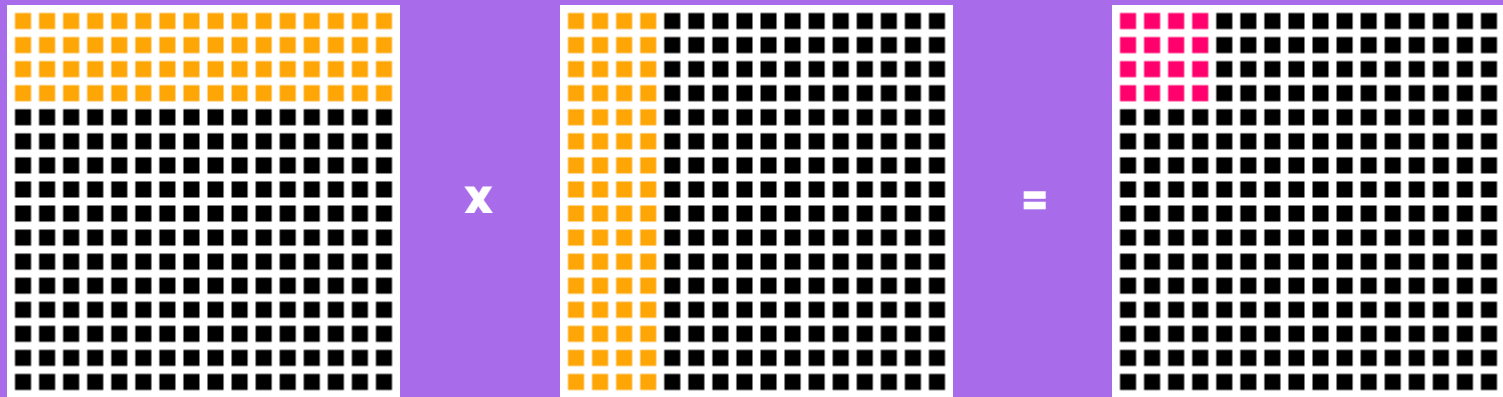


De regulă, când calculăm produsul a două matrice, pentru a calcula o linie a matriciei rezultante luăm o linie a primei matrice și o înmulțim cu fiecare coloană a celei de a doua matrice.

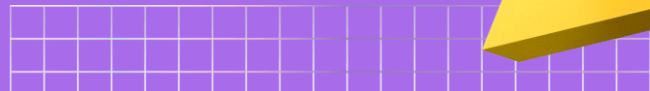




Cache Thrashing/ Pollution & Înmulțirea matricelor

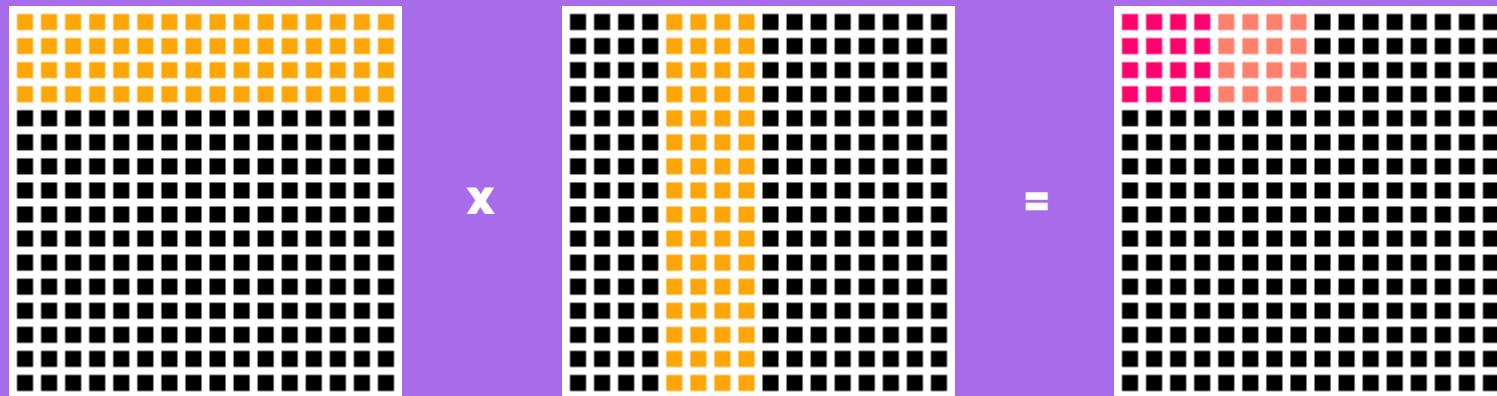


Problema poate fi spartă în block-uri. Pentru a calcula block-ul format din primele 4 linii și 4 coloane din matricea rezultantă este nevoie de primele 4 linii ale primei matrice și primele 4 coloane ale celei de a doua matrice. Dacă acestea au loc în cache, vom îmbunătăți performanța algoritmului.

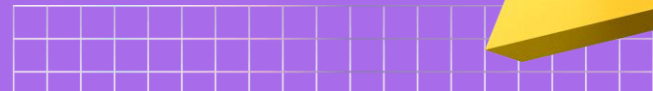




Blocking pentru înmulțirea matricelor



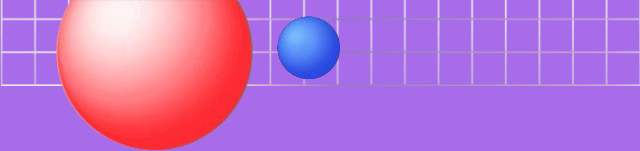
Când terminăm de calculat un block, trecem la următorul block până când întregul produs este calculat. La începutul unui block este posibil să avem **Cache Miss**-uri, dar după ce liniile și coloanele se vor afla în cache, vom avea **Cache Hit**-uri, iar efectuarea operațiilor va fi mai eficientă.



Înmulțirea matricelor v3

```
constexpr auto block_size = 100;

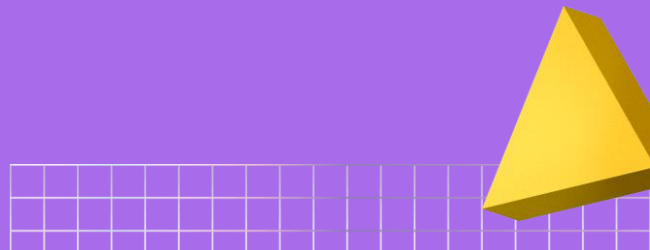
for (auto row = 0; row < size; row += block_size)
{
    for (auto col = 0; col < size; col += block_size)
    {
        for (auto block_row = row; block_row < row + block_size; block_row++)
        {
            for (auto block_col = col; block_col < col + block_size; block_col++)
            {
                c[block_row * size + block_col] = 0.0;
                for (auto dot = 0; dot < size; dot++)
                    c[block_row * size + block_col] += a[block_row * size + dot] *
                                                         b_transposed[block_col * size + dot];
            }
        }
    }
}
```



Fun fact

Considerăm un graf G orientat, neponderat, căruia îi atribuim matricea de adiacență A .

Atunci, elementul de pe poziția (i, j) al matricei A^n va conține numărul de drumuri de lungime n pornind de la nodul i al grafului către nodul j .





**Array-uri locale
statica**



Array-uri de float-uri

```
float look_up_float(int index)
{
    float table[] =
    {
        -0.997497f,0.127171f,-0.613392f,0.617481f,0.170019f,-0.0402539f,-0.299417f,0.791925f,
        0.64568f,0.49321f,-0.651784f,0.717887f,0.421003f,0.0270699f,-0.39201f,-0.970031f,
        -0.817194f,-0.271096f,-0.705374f,-0.668203f,0.97705f,-0.108615f,-0.761834f,-0.990661f,
        -0.982177f,-0.24424f,0.0633259f,0.142369f,0.203528f,0.214332f,-0.667531f,0.32609f,
        -0.0984222f,-0.295755f,-0.885922f,0.215369f,0.566637f,0.605213f,0.0397656f,-0.3961f,
        0.751945f,0.453352f,0.911801f,0.851436f,0.0787072f,-0.715323f,-0.0758385f,-0.529344f,
        0.724479f,-0.580798f,0.559313f,0.687307f,0.993591f,0.99939f,0.222999f,-0.215125f,
        -0.467574f,-0.405438f,0.680288f,-0.952513f,-0.248268f,-0.814753f,0.354411f,-0.88757f
    };

    return table[index];
}
```

VS

```
float look_up_float(int index)
{
    static float table[] =
    {
        -0.997497f,0.127171f,-0.613392f,0.617481f,0.170019f,-0.0402539f,-0.299417f,0.791925f,
        0.64568f,0.49321f,-0.651784f,0.717887f,0.421003f,0.0270699f,-0.39201f,-0.970031f,
        -0.817194f,-0.271096f,-0.705374f,-0.668203f,0.97705f,-0.108615f,-0.761834f,-0.990661f,
        -0.982177f,-0.24424f,0.0633259f,0.142369f,0.203528f,0.214332f,-0.667531f,0.32609f,
        -0.0984222f,-0.295755f,-0.885922f,0.215369f,0.566637f,0.605213f,0.0397656f,-0.3961f,
        0.751945f,0.453352f,0.911801f,0.851436f,0.0787072f,-0.715323f,-0.0758385f,-0.529344f,
        0.724479f,-0.580798f,0.559313f,0.687307f,0.993591f,0.99939f,0.222999f,-0.215125f,
        -0.467574f,-0.405438f,0.680288f,-0.952513f,-0.248268f,-0.814753f,0.354411f,-0.88757f
    };

    return table[index];
}
```

Benchmark

```
constexpr auto runs = 10;
auto total_time = 1LL;
auto total_f = 0.0f;

for (auto run = 0; run < runs; run++)
{
    const auto start_time = clock();

    for (int i = 0; i < 100000000; i++)
        total_f += look_up_float(i & 31);

    const auto finish_time = clock();

    total_time += finish_time - start_time;

    cout << "Time: " << finish_time - start_time << endl;
}

cout << "Average time: " << static_cast<double>(total_time) /
static_cast<double>(runs) << endl;
cout << "Total Float: " << total_f << endl;
```

Rezultate

Array local normal

0.39s

1s

Average time: 397ms

Array local static

0.07s

1s

Average time: 75ms

Array-uri de int-uri

```
int look_up_int(int index)
{
    int table[] =
    {
        1, 7, 4, 0, 9, 4, 8, 8,
        2, 4, 5, 5, 1, 7, 1, 1,
        5, 2, 7, 6, 1, 4, 2, 3,
        2, 2, 1, 6, 8, 5, 7, 6
    };

    return table[index];
}
```

vs

```
int look_up_int(int index)
{
    static int table[] =
    {
        1, 7, 4, 0, 9, 4, 8, 8,
        2, 4, 5, 5, 1, 7, 1, 1,
        5, 2, 7, 6, 1, 4, 2, 3,
        2, 2, 1, 6, 8, 5, 7, 6
    };

    return table[index];
}
```


Rezultate

Array local normal



Average time: 215.6ms

Array local static



Average time: 26.5ms

Benchmark-ul este același folosit în cazul funcției `look_up_float`, adaptat pentru funcția `look_up_int`, codul nu a fost inclus în prezentare.

String-uri

```
char look_up_string(int index)
{
    char str[] = "Miscarea e un paradox temporal."; // 32 bytes string including \0

    return str[index];
}
```

VS

```
char look_up_string(int index)
{
    static char str[] = "Miscarea e un paradox temporal.";

    return str[index];
}
```

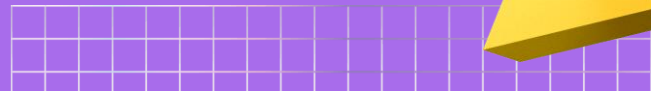


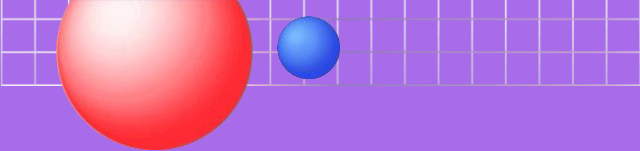
Rezultate

Aprox. 35ms ambele variante.

De ce?

Benchmark-ul este același folosit în cazul funcției `look_up_float`, adaptat pentru funcția `look_up_string`, codul nu a fost inclus în prezentare.

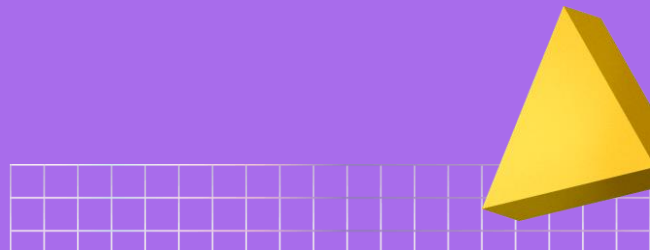


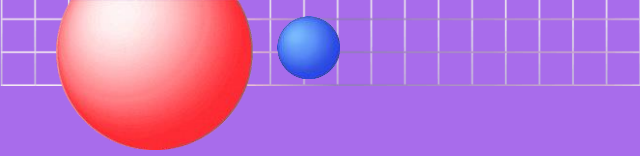


Array-uri locale statice

În cazul array-urilor locale normale, datele sunt copiate pe stivă înainte de a fi accesate.

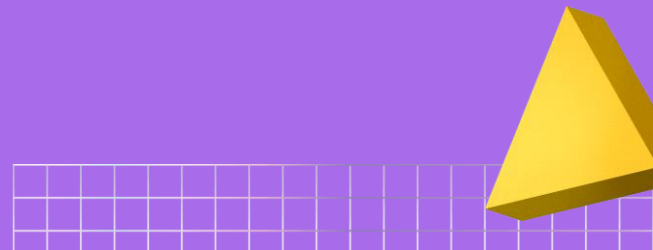
În cazul array-urilor statice, datele se află în permanență undeva în memorie.





String-uri

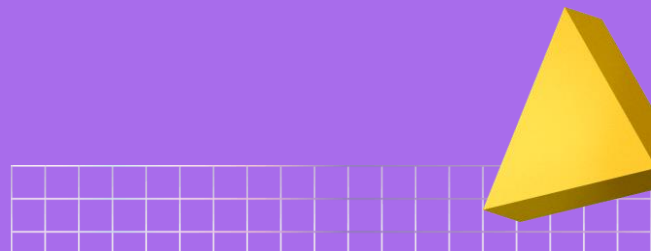
În cazul String-urilor nu există diferențe majore, deoarece string-urile constante sunt salvate în permanență în memorie undeva global.





Warning

Folosiți această tehnică doar în cazul lookup table-urilor sau array-urilor care nu își schimbă valorile!





Unrolling Loops & Dependency Chains



Suma elementelor dintr-un array

```
double sum(const double* doubles, const unsigned int count)
{
    double total = 0.0;

    for (auto i = 0; i < count; i++)
        total += doubles[i];

    return total;
}
```


Suma elementelor dintr-un array v2

```
double sum_unrolled(double* doubles, const unsigned int count)
{
    double total1 = 0.0;
    double total2 = 0.0;
    double total3 = 0.0;
    double total4 = 0.0;

    double* data = doubles;
    for (auto i = 0; i < count / 4; i++)
    {
        total1 += data[0];
        total2 += data[1];
        total3 += data[2];
        total4 += data[3];

        data += 4;
    }

    for (auto i = 0; i < (count & 3); i++)
        total1 += *data++;

    return (total1 + total2) + (total3 + total4);
}
```

Benchmark

```
constexpr auto count    = 100000;
constexpr auto runs     = 20000;
const      auto doubles = new double[count];

for (auto i = 0; i < count; i++)
    doubles[i] = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);

auto start_time = clock();

auto sum1 = 0.0;
for (auto run = 0; run < runs; run++)
    sum1 = sum(doubles, count);

auto finish_time = clock();

cout << "Regular Sum (" << sum1 << ") Time: " << finish_time - start_time << "ms." << endl;

start_time = clock();

auto sum2 = 0.0;
for (auto run = 0; run < runs; run++)
    sum2 = sum_unrolled(doubles, count);

finish_time = clock();

cout << "Unrolled Sum (" << sum2 << ") Time: " << finish_time - start_time << "ms." << endl;
```

Rezultate

Suma normală

v2

1.5s

2s

0.38s

2s

Timp: 1506ms

Timp: 386ms

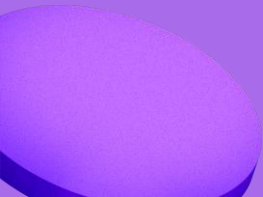
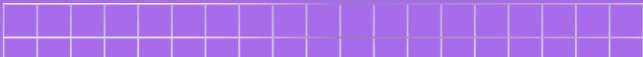


Loop Unrolling

Procesorul **NU** ia câte o singură instrucțiune pe care o execută și apoi trece la următoarea

Instrucțiunile ajung în niște buffere și cozi ale procesorului înainte de a fi executate.

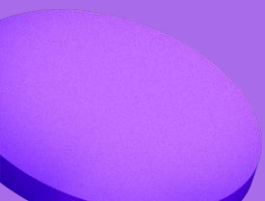
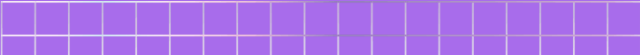
Pentru ca o instrucțiune să poată fi executată, este nevoie ca toate resursele de care depinde aceasta să fie disponibile și să existe un pipe/port disponibil pe procesor care să o poată executa.





Loop Unrolling

Dacă două sau mai multe instrucțiuni au disponibile toate resursele necesare și există pipe-uri/porturi disponibile pentru acestea, ele se vor executa în paralel!



Suma elementelor dintr-un array

```
double sum(const double* doubles, const unsigned int count)
{
    double total = 0.0;

    for (auto i = 0; i < count; i++)
        total += doubles[i];

    return total;
}
```

La fiecare iterație, instrucțiunea `total += doubles[i];` depinde de valoarea lui `total` de la iterația anterioară

Suma elementelor dintr-un array v2

```
double sum_unrolled(double* doubles, const unsigned int count)
```

```
{
```

```
    double total1 = 0.0;
```

```
    double total2 = 0.0;
```

```
    double total3 = 0.0;
```

```
    double total4 = 0.0;
```

```
    double* data = doubles;
```

```
    for (auto i = 0; i < count / 4; i++)
```

```
    {
```

```
        total1 += data[0];
```

```
        total2 += data[1];
```

```
        total3 += data[2];
```

```
        total4 += data[3];
```

```
        data += 4;
```

```
    }
```

```
    for (auto i = 0; i < (count & 3); i++)
```

```
        total1 += *data++;
```

```
    return (total1 + total2) + (total3 + total4); // parentheses are for breaking a
```

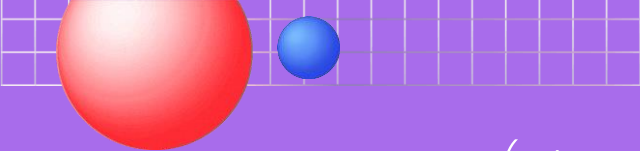
```
        // dependency chain here too.
```

```
}
```

Instrucțiunile

total1 += data[0]; total2 += data[1];
total3 += data[2]; total4 += data[3];

sunt independente, deci se pot
executa simultan dacă există 4
porturi pentru operații aritmetice
disponibile



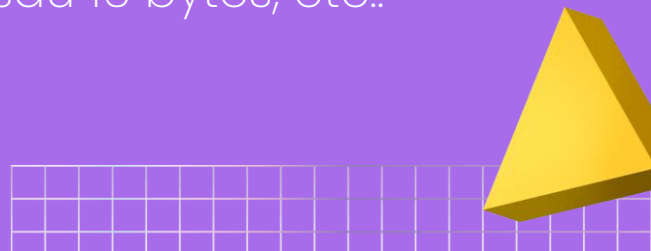
SIMD

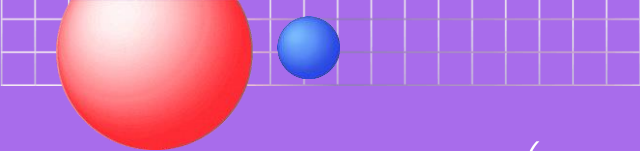
(Single Instruction Multiple Data)

Procesorul conține tipuri speciale de regiștrii care pot lucra cu mai multe date simultan.

Acești regiștrii pot avea diverse dimensiuni (128, 256 sau 512 biți) și pot conține mai multe valori.

De exemplu: un registru pe 256 de biți poate fi folosit pentru a lucra cu 4 double-uri sau 8 float-uri. Un registru pe 128 de biți poate fi folosit pentru a lucra cu 4 float-uri sau 16 bytes, etc..



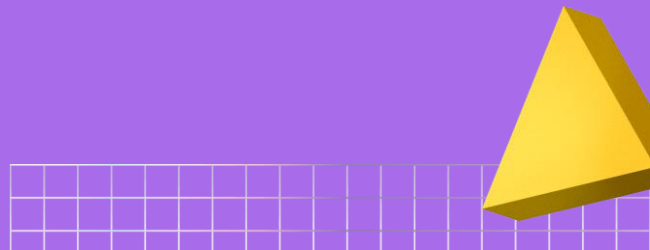


SIMD

(Single Instruction Multiple Data)

Există instrucțiuni speciale care folosesc acești regiștrii pentru a efectua mai multe calcule simultan.

Aceste instrucțiuni pot fi folosite în assembly, dar le putem folosi și în C/C++ prin intermediul intrinsecilor.



Suma elementelor dintr-un array v3

```
double sum_avx(double* doubles, const unsigned int count)
{
    auto total = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);

    double* data = doubles;
    for (auto i = 0; i < count / 4; i++)
    {
        total = _mm256_add_pd(total, _mm256_load_pd(data));
        data += 4;
    }

    total = _mm256_hadd_pd(total, total);
    const auto total2 = _mm256_permute2f128_pd(total, total, 1);
    total = _mm256_add_pd(total, total2);

    for (auto i = 0; i < (count & 3); i++)
        total = _mm256_add_pd(total, _mm256_set_pd(0.0, 0.0, 0.0, *data++));

    double result[4];
    _mm256_store_pd(result, total);
    return result[0];
}
```



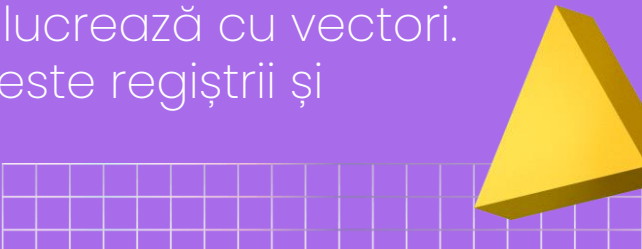
SIMD

(Single Instruction Multiple Data)

Variabila total ne-o putem imagina ca pe un registru de tip SIMD care păstrează 4 double-uri. Acele double-uri țin locul celor 4 acumulatori din varianta a doua a algoritmului.

Uneori, compilatoarele optimizează loop-urile folosind metodele explorate anterior. Totuși, nu întotdeauna optimizările sugerate de compilator sunt optime.

Altă utilitate SIMD este în cazul operațiilor matematice cu vectori și matrice. De regulă, engine-urile și framework-urile de jocuri dispun de câte o librărie de matematică care lucrează cu vectori. Aceste librării de regulă sunt wrappere peste regiștrii și instrucțiunile SIMD.



Suma elementelor dintr-un array v4

```
double sum_axv_unrolled(double* doubles, const unsigned int count)
{
    auto total1 = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
    auto total2 = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
    auto total3 = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
    auto total4 = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);

    double* data = doubles;
    for (auto i = 0; i < count / 16; i++)
    {
        total1 = _mm256_add_pd(total1, _mm256_load_pd(&data[0]));
        total2 = _mm256_add_pd(total2, _mm256_load_pd(&data[4]));
        total3 = _mm256_add_pd(total3, _mm256_load_pd(&data[8]));
        total4 = _mm256_add_pd(total4, _mm256_load_pd(&data[12]));

        data += 16;
    }

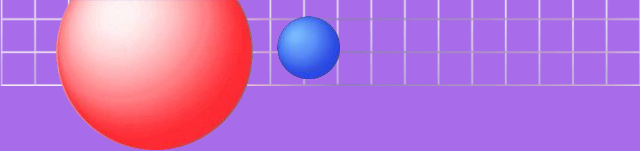
    total1 = _mm256_add_pd(total1, total2);
    total3 = _mm256_add_pd(total3, total4);

    total1 = _mm256_add_pd(total1, total3);

    total1 = _mm256_hadd_pd(total1, total1);
    const auto tmp = _mm256_permute2f128_pd(total1, total1, 1);
    total1 = _mm256_add_pd(total1, tmp);

    for (auto i = 0; i < (count & 15); i++)
        total1 = _mm256_add_pd(total1, _mm256_set_pd(0.0, 0.0, 0.0, *data++));

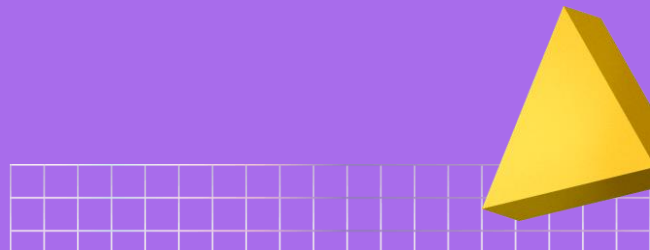
    double result[4];
    _mm256_store_pd(result, total1);
    return result[0];
}
```



Loop Unrolling

Când facem loop unrolling trebuie să estimăm numărul de pipe-uri pe care le avem disponibile pentru a ne executa instrucțiunile simultan.

De la un anumit punct, desfășurarea loop-urilor nu mai îmbunătățește semnificativ performanța,
so **don't overdo it!!!**



Abaterea standard

```
float standard_deviation(const float* data, const unsigned int count)
{
    auto total = 0.0f;

    for (auto i = 0; i < count; i++)
        total += data[i];

    const float average = total / static_cast<float>(count);
    total = 0.0f;

    for (auto i = 0; i < count; i++)
        total += (data[i] - average) * (data[i] - average);

    total /= static_cast<float>(count);

    return sqrtf(total);
}
```

Abaterea standard v2

```
float standard_deviation_one_pass(const float* data, const unsigned int count)
{
    auto s1 = 0.0f;
    auto s2 = 0.0f;
    auto n   = static_cast<float>(count);

    for (auto i = 0; i < count; i++)
    {
        s1 += data[i];
        s2 += data[i] * data[i];
    }

    return sqrtf(n * s2 - s1 * s1) / n;
}
```

Abaterea standard v3

```
float standard_deviation_one_pass_unrolled(const float* data, const unsigned int count)
{
    const auto n = static_cast<float>(count);

    auto s1_1 = 0.0f;
    auto s1_2 = 0.0f;

    auto s2_1 = 0.0f;
    auto s2_2 = 0.0f;

    for (auto i = 0; i + 1 < count; i += 2)
    {
        s1_1 += data[i];
        s1_2 += data[i + 1];

        s2_1 += data[i] * data[i];
        s2_2 += data[i + 1] * data[i + 1];
    }

    if ((count & 1) == 1)
    {
        s1_1 += data[count - 1];
        s2_1 += data[count - 1] * data[count - 1];
    }

    const auto s1 = s1_1 + s1_2;
    const auto s2 = s2_1 + s2_2;

    return sqrtf(n * s2 - s1 * s1) / n;
}
```


Abaterea standard v4

```
float standard_deviation_one_pass_unrolled_avx(float* d,
const unsigned int count)
{
    const auto n = static_cast<float>(count);

    auto s1_1 = _mm256_set_ps(0.0f, 0.0f, 0.0f, 0.0f,
                               0.0f, 0.0f, 0.0f, 0.0f);
    auto s1_2 = _mm256_set_ps(0.0f, 0.0f, 0.0f, 0.0f,
                               0.0f, 0.0f, 0.0f, 0.0f);

    auto s2_1 = _mm256_set_ps(0.0f, 0.0f, 0.0f, 0.0f,
                               0.0f, 0.0f, 0.0f, 0.0f);
    auto s2_2 = _mm256_set_ps(0.0f, 0.0f, 0.0f, 0.0f,
                               0.0f, 0.0f, 0.0f, 0.0f);

    float* data = d;
    for (auto i = 0; i < count / 16; i++)
    {
        const auto data0 = _mm256_load_ps(&data[0]);
        const auto data1 = _mm256_load_ps(&data[8]);

        s1_1 = _mm256_add_ps(s1_1, data0);
        s1_2 = _mm256_add_ps(s1_2, data1);

        s2_1 = _mm256_add_ps(s2_1, _mm256_mul_ps(data0, data0));
        s2_2 = _mm256_add_ps(s2_2, _mm256_mul_ps(data1, data1));

        data += 16;
    }
}
```

```
s1_1 = _mm256_add_ps(s1_1, s1_2);
s2_1 = _mm256_add_ps(s2_1, s2_2);

s1_1 = _mm256_hadd_ps(s1_1, s1_1);
s2_1 = _mm256_hadd_ps(s2_1, s2_1);
s1_1 = _mm256_hadd_ps(s1_1, s1_1);
s2_1 = _mm256_hadd_ps(s2_1, s2_1);

const auto tmp1 = _mm256_permute2f128_ps(s1_1, s1_1, 1);
const auto tmp2 = _mm256_permute2f128_ps(s2_1, s2_1, 1);

s1_1 = _mm256_add_ps(s1_1, tmp1);
s2_1 = _mm256_add_ps(s2_1, tmp2);

for (auto i = 0; i < (count & 15); i++)
{
    const auto value = *data++;
    s1_1 = _mm256_add_ps(s1_1, _mm256_set_ps(0.0f, 0.0f, 0.0f, 0.0f,
                                              0.0f, 0.0f, 0.0f, value));

    s2_1 = _mm256_add_ps(s2_1, _mm256_set_ps(0.0f, 0.0f, 0.0f, 0.0f,
                                              0.0f, 0.0f, 0.0f, value * value));
}

float result1[8], result2[8];
_mm256_store_ps(result1, s1_1);
_mm256_store_ps(result2, s2_1);

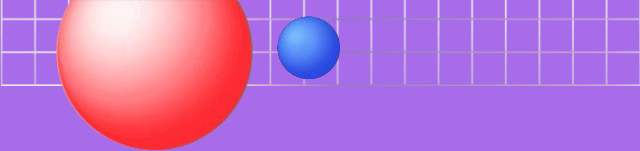
const auto s1 = result1[0];
const auto s2 = result2[0];

return sqrtf(n * s2 - s1 * s1) / n;
}
```



Branchless Programming

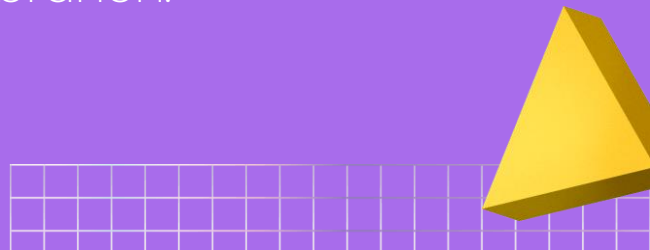


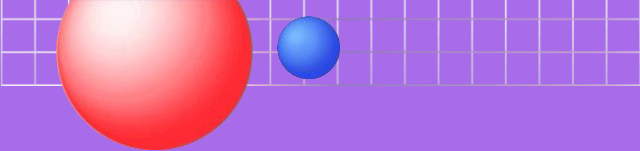


Branch-uri

Spunem că avem branching oricând programul este nevoit să treacă de la o zonă de instrucțiuni la alta. De regulă avem branching în cazul if-urilor și al loop-urilor.

Când se pregătesc instrucțiunile pe care procesorul urmează să le execute, iar procesorul întâlnește o instrucțiune de tip branch, acesta nu știe ce path va urma să execute, și prin urmare, nici ce instrucțiuni să încarce după acel branch.





Branch-uri

Procesorul va încerca să ghicească instrucțiunile care vor urma după branch-ul respectiv folosind niște algoritmi probabiliști simplii. Acest lucru se numește **Branch Prediction**.

În cazul în care procesorul a ghicit greșit path-ul, acesta va fi nevoit să elimine instrucțiunile încărcate degeaba și să le încarce pe cele bune. Acest lucru este lent.

Instrucțiunile de tip Branch sunt cele mai lente instrucțiuni.

Ne dorim să minimizăm folosirea branch-urilor acolo unde este nevoie de performanță.



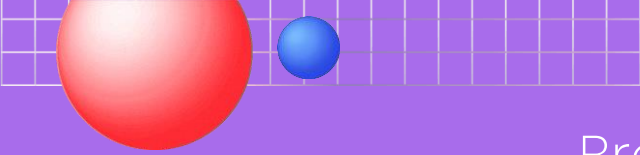
Minim

```
// We assume that a and b have an  
// uniform distribution.
```

```
int min(const int a, const int b)  
{  
    if (a < b)  
        return a; // 50% chance  
    else  
        return b; // 50% chance  
}
```

Minim v2

```
int min_branchless(const int a,  
                   const int b)  
{  
    return a * (a < b) +  
           b * (b <= a); // no if statement  
}
```

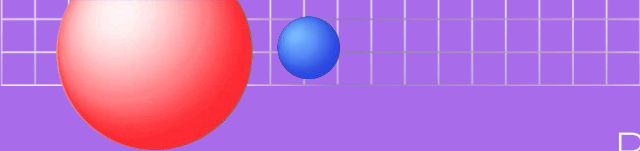


Branchless Programming

Compilatoarele folosesc uneori aceasta tehnică pentru a optimiza codul rezultat.

Nu mereu reuşesc compilatoarele să facă aceste optimizări. În cazul în care compilatorul nu face automat aceste optimizări, ele trebuie făcute manual. Pentru a vedea ce optimizări a făcut compilatorul trebuie inspectat codul assembly generat.

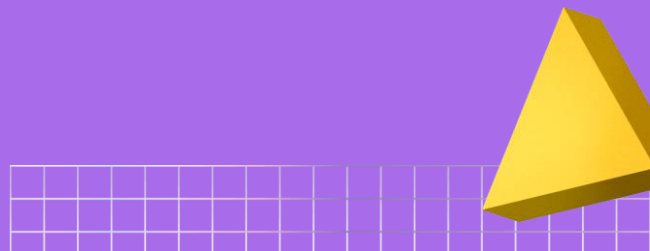




Branchless Programming

În cazul funcției definite anterior, un compilator bun ar optimiza direct funcția, renunțând la branch-uri.

Probabil codul rezultat ar fi chiar mai eficient decât versiunea compilată a funcției branchless definită anterior.



To Upper

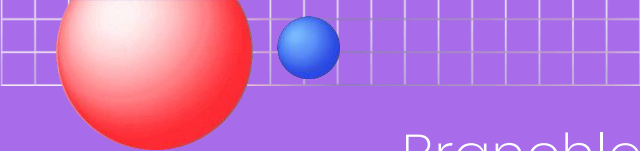
```
void to_upper(char* d, const int count)
{
    for (auto i = 0; i < count; i++)
    {
        if (d[i] >= 'a' && d[i] <= 'z')
            d[i] -= 32; // 'a' - 'A' = 32
    }
}
```

To Upper v2

```
void to_upper_branchless(char* d, const int count)
{
    for (auto i = 0; i < count; i++)
    {
        d[i] = d[i] * !(d[i] >= 'a' && d[i] <= 'z') +
                (d[i] - 32) * (d[i] >= 'a' && d[i] <= 'z');
    }
}
```

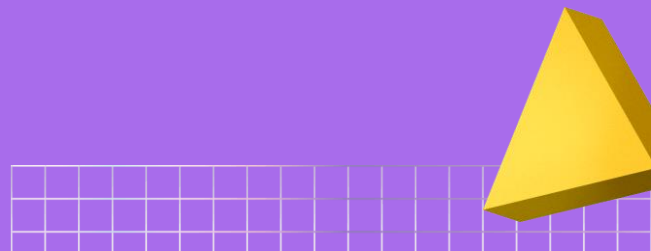
To Upper v3

```
void to_upper_branchless_2(char* d,  
    const int count)  
{  
    for (auto i = 0; i < count; i++)  
        d[i] -= 32 * (d[i] >= 'a' && d[i] <= 'z');  
}
```



Branchless Programming folosind SIMD

Putem combina combina această tehnică de eliminare a branch-urilor cu SIMD pentru a obține performanțe bune.



To Upper v4

```
void to_upper_branchelss_avx(char* d, const int count)
{
    auto ones      = _mm256_set1_epi8(-1);

    auto thirty_two = _mm256_set1_epi8(32);
    auto a_minus_one = _mm256_set1_epi8('a' - 1);
    auto z           = _mm256_set1_epi8('z');

    char* data = d;
    for (auto i = 0; i < count / 32; i++)
    {
        auto chars      = _mm256_loadu_epi8(data);           // d[i] (we actually refer to the first 32 characters starting at the memory
                                                                // address d. Writing it as d[i] just makes the following comments easier to
                                                                // understand intuitively.)

        auto compare_a   = _mm256_cmpgt_epi8(chars, a_minus_one); // d[i] >= 'a' (for a given byte, if its value is greater than the
                                                                // corresponding byte in the second register, all the resulting bits of the
                                                                // mask for that given byte will be equal to 1, otherwise they will be equal
                                                                // to 0)

        auto compare_z   = _mm256_cmpgt_epi8(chars, z);         // d[i] > 'z'
        auto compare_z    = _mm256_xor_epi32(compare_z, ones);   // d[i] <= 'z'

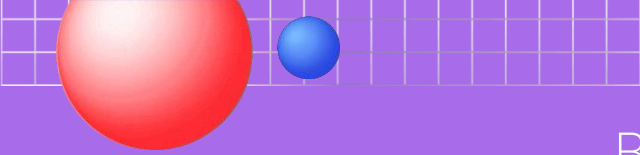
        auto compare     = _mm256_and_epi32(compare_a, compare_z); // mask for d[i] >= 'a' && d[i] <= 'z'
        auto lower       = chars;
        auto compare_complement = _mm256_xor_epi32(compare, ones); // mask for !(d[i] >= 'a' && d[i] <= 'z')

        chars            = _mm256_and_epi32(chars, compare_complement); // keep only the upper characters here
        lower            = _mm256_sub_epi8(lower, thirty_two);           // upper the chracters which were originally lower
        lower            = _mm256_and_epi32(lower, compare);             // keep only the characters that were originally lower here

        chars            = _mm256_or_epi32(chars, lower);             // merge all characters

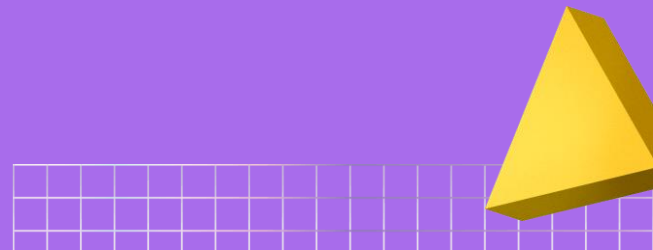
        _mm256_storeu_epi8(data, chars);                             // save the results into memory

        data += 32;
    }
}
```



Branchless Programming

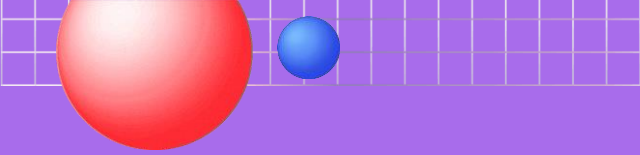
Pentru atribuiri condiționate, atât procesoarele cât și plăcile video au instrucțiuni dedicate.





Multumesc!





Bibliografie/ Resurse

Creel Youtube Channel

<https://www.youtube.com/@WhatsACreel>

Agner Fog's Optimization Manuals

<https://www.agner.org/optimize/#manuals>

