

Санах ойн шатлал

F.CS306 ПАРАЛЛЕЛ ПРОГРАММЧЛАЛ – Лекц 5

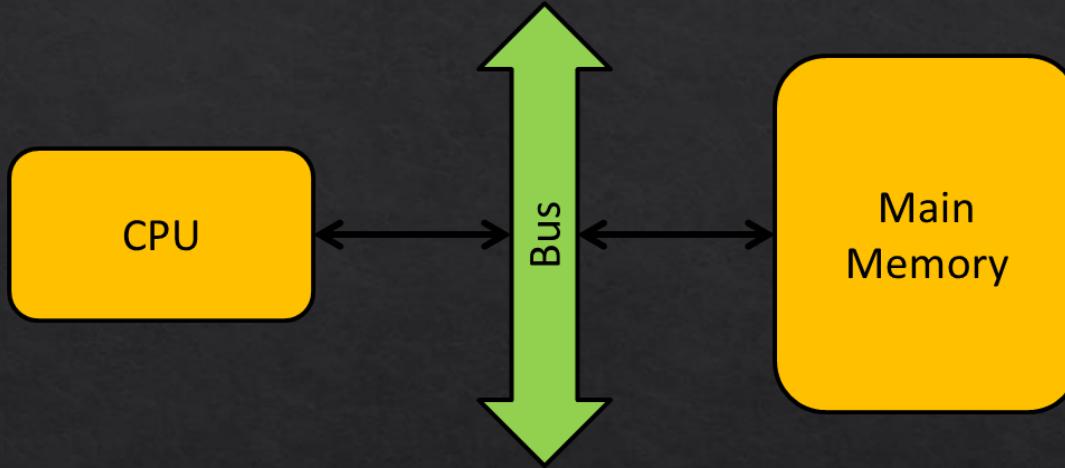
Г.ГАНБАТ

ganbatg@must.edu.mn

Хичээлийн агуулга

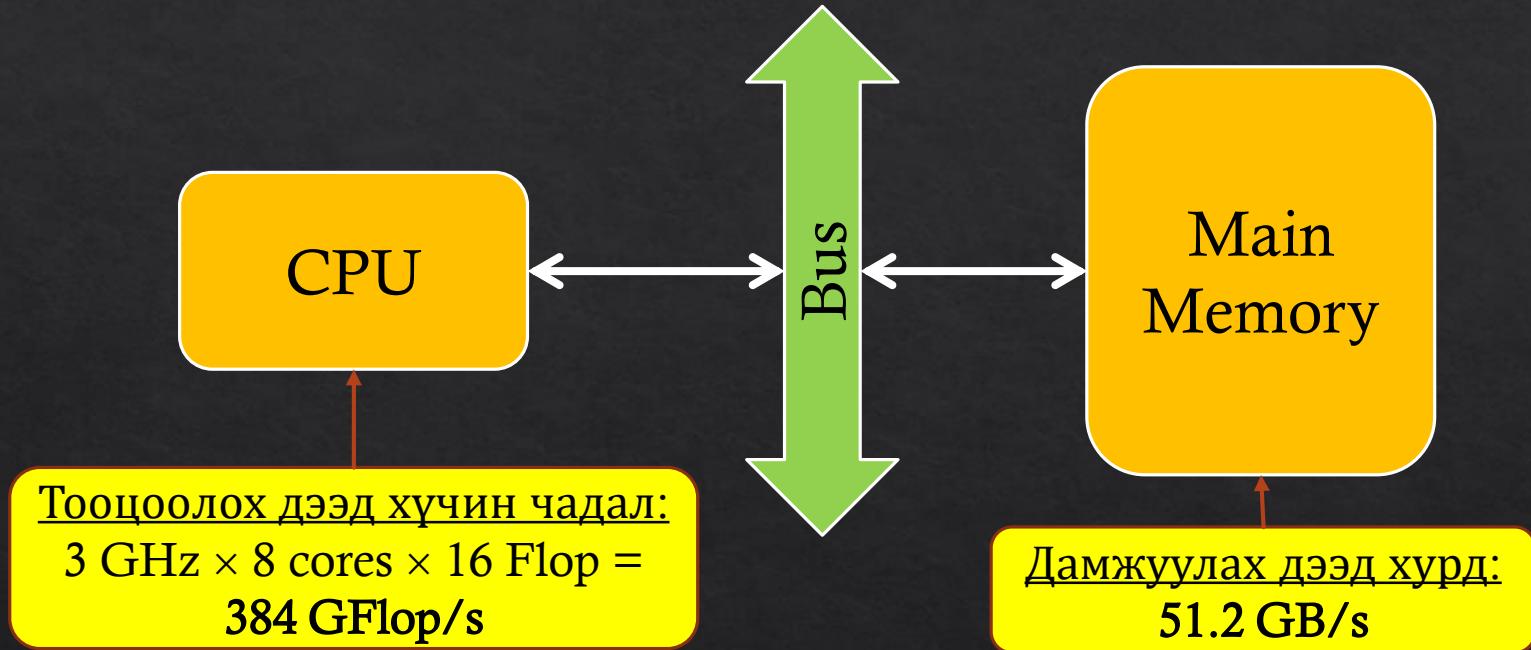
- ❖ Өнөө үед нэг урсгалтай CPU-ийн чадамж ашиггүй болсон
- ❖ Орчин үеийн архитектурын давуу талыг гүйцэд ашиглахад параллел алгоритмын зохиомжоос гадна **санах ойн систем** болон **вектор нэгжийн** зэрэг боломжуудын мэдлэг шаардлагатай
- ❖ CPU болон үндсэн санах ойн дунд *von Neumann bottleneck*-ийг багасгахаар байрлуулсан шуурхай кэшүүд бүхий санах ойн шатлалын тухай авч үзнэ
- ❖ Идэвхтэй санах ойн системийн үр дүнтэй хэрэглээг бий болгосон програмуудыг бичих
- ❖ Зэрэгцээ цөмт CPU систем дэх Cache Coherency болон False Sharing –г ойлгох

Сонгодог Фон Нейман Архитектур



- ❖ Компьютерийн системийн өмнөх үед үндсэн санах ойд хандах хугацаа болон тооцоолол боловсруулах хугацаа хангалттай тэнцвэртэй байсан.
- ❖ Сүүлийн үед тооцооллын хурд үндсэн санах ойн хандалтын хурдтай харьцуулахад асар хурдтай өссөн нь чадавхийн ихээхэн зөрүүг бий болгосон
- ❖ von Neumann Bottleneck: CPU тооцоолох болон үндсэн санах ойн (DRAM) хурднуудын зөрүүтэй байдал

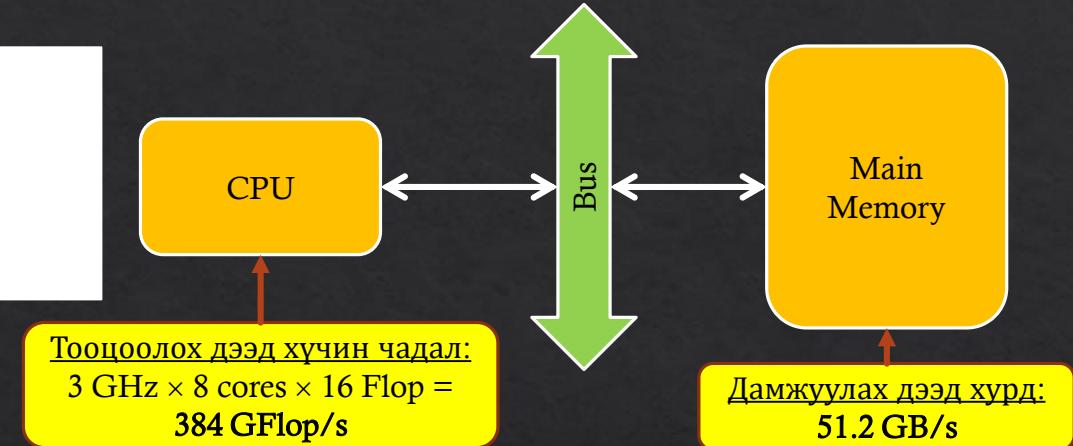
Фон Нейман Bottleneck – Жишээ



- ❖ Үндсэн санах ойд хадгалагдсан n ширхэг double-precision тоонуудыг агуулсан u, v гэсэн хоёр веторын dot product-г тооцоолох **хүчин чадлын дээд хязгаар** тогтоох зорилгоор хялбаршуулсан загвар

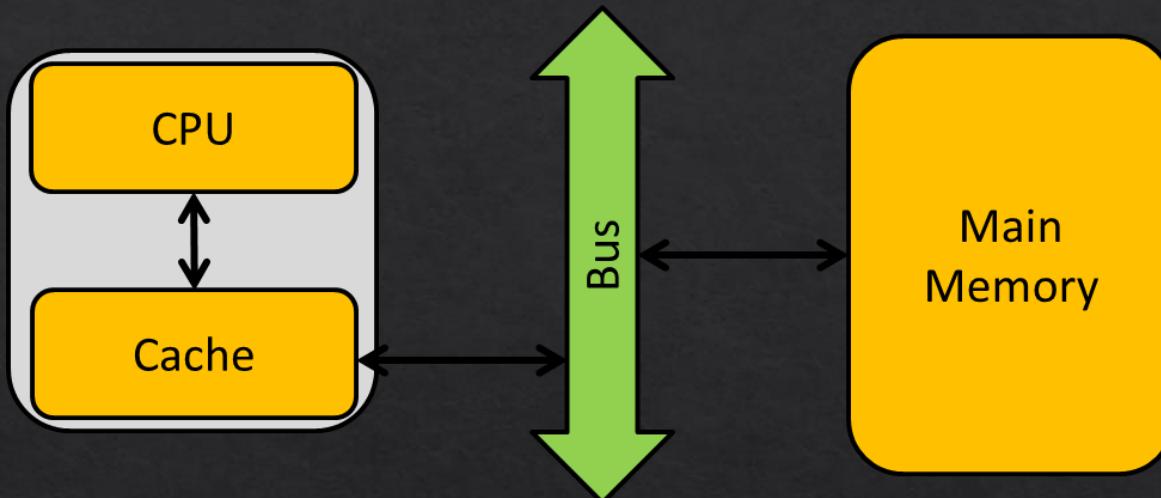
DOT –ЫН ГҮЙЦЭТГЭЛ

```
// Dot Product  
double dotp = 0.0;  
for (int i = 0; i<n; i++)  
    dotp += u[i] * v[i];
```



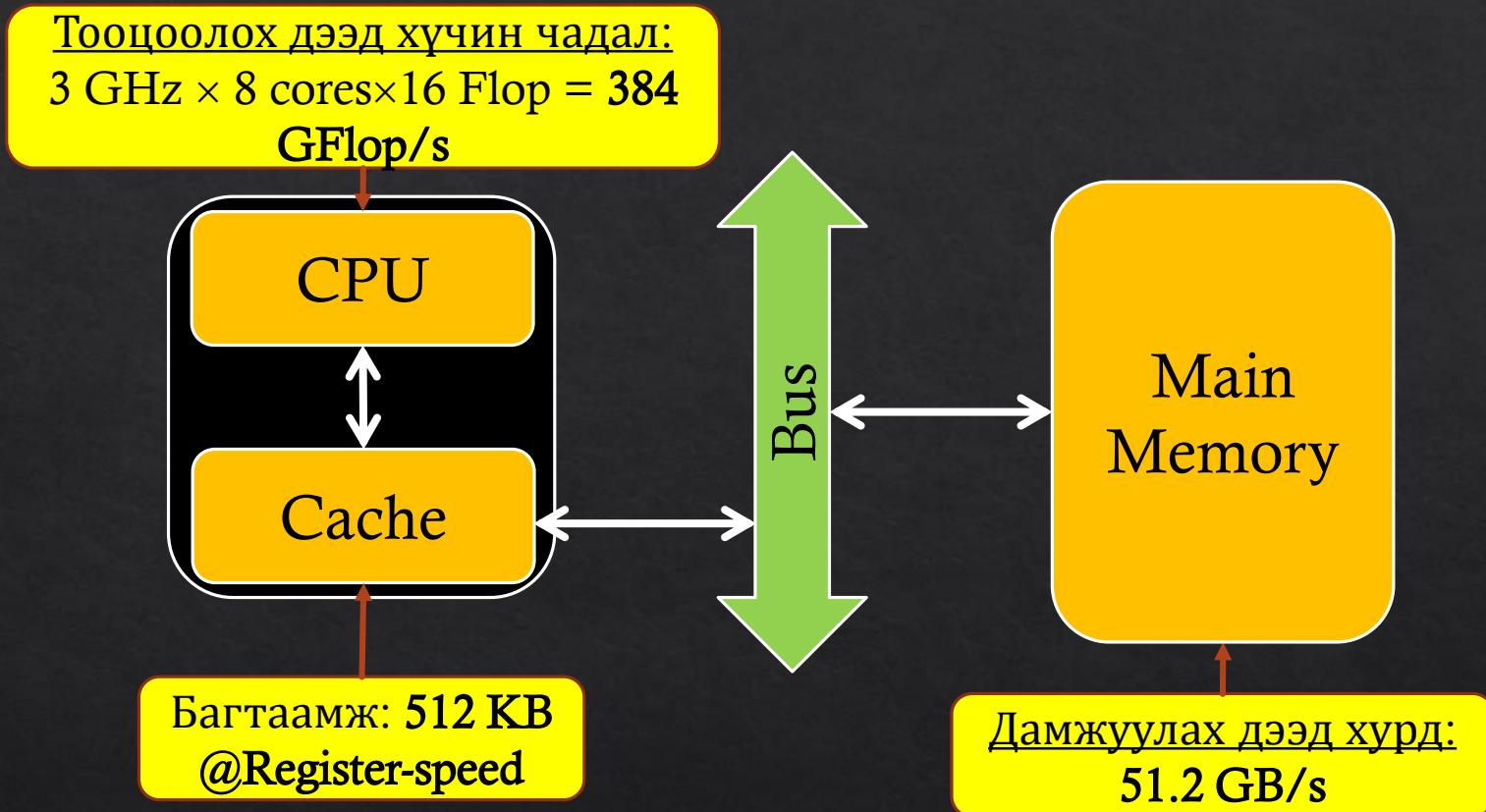
- ❖ Жишээ: $n = 2^{30}$
- ❖ Тооцоолох хугацаа: $t_{\text{comp}} = \frac{2 \text{ GFlop}}{384 \text{ GFlop/s}} = 5.2 \text{ ms}$
 - ❖ Үйлдлийн тоо: $2 \cdot n = 2^{31} \text{ Flops} = 2 \text{ GFlops}$
- ❖ Өгөгдөл зөөх хугацаа: $t_{\text{mem}} = \frac{16 \text{ GB}}{51.2 \text{ GB/s}} = 312.5 \text{ ms}$
 - ❖ Зөөх өгөгдлийн хэмжээ: $2 \cdot 2^{30} \cdot 8 \text{ B} = 16 \text{ GB}$
- ❖ Ажиллах хугацаа : $t_{\text{exec}} \geq \max(5.2ms, 312.5ms) = 312.5ms$
 - ❖ Боломжит хүчин чадал: $\frac{2 \text{ GFlop}}{312.5 \text{ ms}} = 6.4 \text{ GFlop/s} (<2\% \text{ of peak})$
- ❖ ⇒ Dot product нь санах ойд хязгаарлагдана(өгөгдлийн дахин ашиглалтгүй)

Нэг кэштэй CPU-ны үндсэн бүтэц



- ❖ CPU нь ихэвчлэн кэшийн гурван түвшний шаталалтай(L1, L2, L3)
 - ❖ Одоогийн CUDA-enabled GPU нь хоёр түвшинтэй
- ❖ Үндсэн санах ойтой харьцуулахад bandwidth өндөр, алдагдал багатай боловч маш бага багтаамжтой
- ❖ Хурд болон багтаамж урвуу харьцаатай
 - ❖ Жнь: L1-кэш жижиг боловч хурдан, L3-кэш харьцангуй том ч удаан.
- ❖ Кэш нь нэг цөмийн эзэмшилд эсвэл олон цөм дундын байж болно

Кэш санах ой – Жишээ

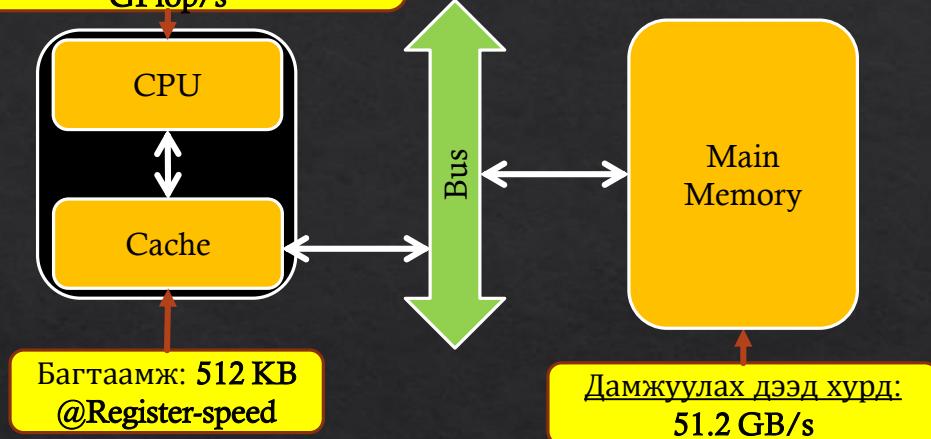


- ❖ Үндсэн санах ойд хадгалагдсан тус бүр нь $n \times n$ хэмжээтэй $W = U \times V$ матрицийн үржвэрийг тооцоолох **хүчин чадлын дээд хязгаар** тогтоох зорилгоор хялбаршуулсан загвар

Матрицын үржвэрийн гүйцэтгэл

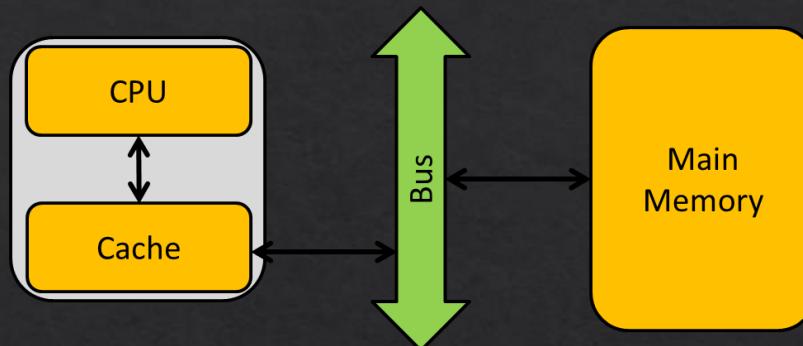
```
//Matrix Multiplication
for (int i = 0; i<n; i++)
    for (int j = 0; j < n; j++) {
        double dotp = 0;
        for (int k = 0; k<n; k++)
            dotp += U[i][k]*V[k][j];
        W[i][j] = dotp;
    }
```

Тооцоолох дээд хүчин чадал:
3 GHz × 8 cores×16 Flop = 384
GFlop/s



- ❖ Жишээ: $n = 128$
- ❖ Өгөгдөл зөөх хугацаа: $t_{\text{mem}} = \frac{384 \text{ KB}}{51.2 \text{ GB/s}} = 7.5 \mu\text{s}$
 - ❖ Зөөх өгөгдлийн хэмжээ (Кэшээс, кзшрүү): $n = 128: 128^2 \times 3 \times 8\text{B} = 384 \text{ KB}$ (Кэшд багтаж)
- ❖ Тооцоолох хугацаа: $t_{\text{comp}} = \frac{2^{22} \text{ Flop}}{384 \text{ GFlop/s}} = 10.4 \mu\text{s}$
 - ❖ Үйлдлийн тоо: $2 \cdot n^3 = 2 \cdot 128^3 = 2^{22} \text{ Flops}$
- ❖ Ажиллах хугацаа: $t_{\text{exec}} \geq 7.5 \mu\text{s} + 10.4 \mu\text{s} = 17.9 \mu\text{s}$
 - ❖ Боломжит хүчин чадал : $\frac{2^{22} \text{ Flop}}{17.9 \mu\text{s}} = 223 \text{ GFlop/s}$ ($\approx 60\%$ of peak)
- ❖ ⇒ ММ нь дахин ашиглах өгөгдөл ихтэй! Матрицийн хэмжээ кэшээс том бол яах вэ?

Кэш алгоритмууд



Үндсэн санах ойгоос ямар өгөгдлийг ачаалах вэ?

Кэш дотор хаана хадгалах вэ?

Кэш дүүрсэн бол аль өгөгдлийг хасах вэ?

- ❖ Кэшийг хэрэглэгч шууд удирдаад байх шаардлагагүй
- ❖ Кэш удирдлагын (*кэш алгоритмууд*) багц нь програм ажиллах явцад аль өгөгдөл кэшлэгдэхийг тодорхойлдог
- ❖ **Cache hit:** Өгөгдлийн хүсэлт нь үндсэн санах ойгоос татахгүйгээр кэшээс уншуулан үйлчлүүлж болно.
- ❖ **Cache miss:** Эсрэгээрээ/үйлчлүүлэгхгүй
- ❖ **Hit ratio:** cache hit-ийн өгөгдлийн хүсэлтэнд хариулах хувь

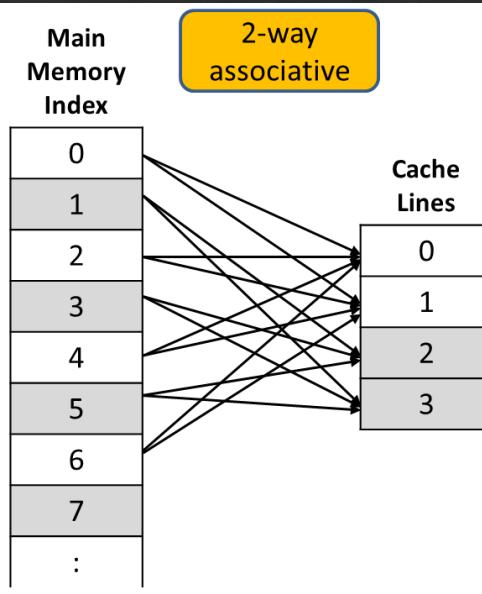
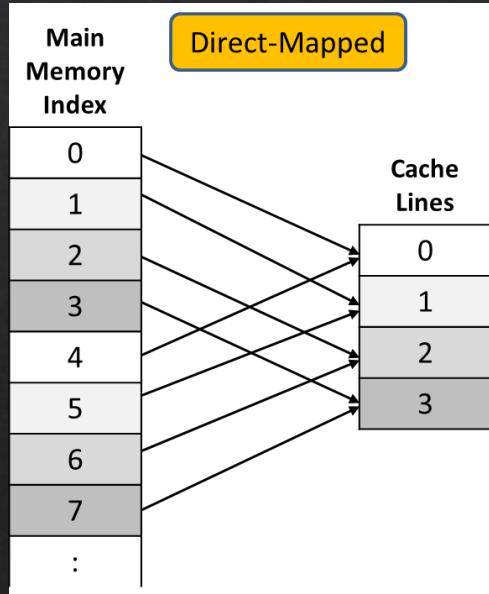
Кэшлэлтийн алгоритм - Spatial Locality

Үндсэн санах ойгоос ямар өгөгдлийг ачаалах вэ?

```
//maximum of an array (elements stored contiguously)
for (i = 0; i<n; i++)
    maximum = max(a[i], maximum);
```

- ❖ Cache Line: санах ойн нэг байршилтай хэд хэдэн мэдээлэл
- ❖ Зөвхөн нэг утга шаардахын оронд бүтэн кэш шугаманд санах ойн зэргэлдээ хаягуудаас утгуудыг ачаалдаг.
- ❖ Жишээ: 64 В хэмжээтэй кэш шугам болон double precision утга
 - ❖ Давталт эхний алхам: $a[0]$ дуудахад Cache miss болно
 - ❖ $a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7]$ гэсэн 8 дараалсан утга нэг кэш шугам дээр ачаалагдана.
 - ❖ Дараагийн 7 алхамд Cache hit ажиллалана
 - ❖ $a[8]$ гэсэн дараагийн хүсэлт Cache miss болно, гэх мэт
 - ❖ Манай жишээний хувьд Hit ratio 87.5%-тай өндөр байна.

Кэшлэлийн алгоритм – Temporal Locality



Кэшийг хаана хадгалдаг вэ?

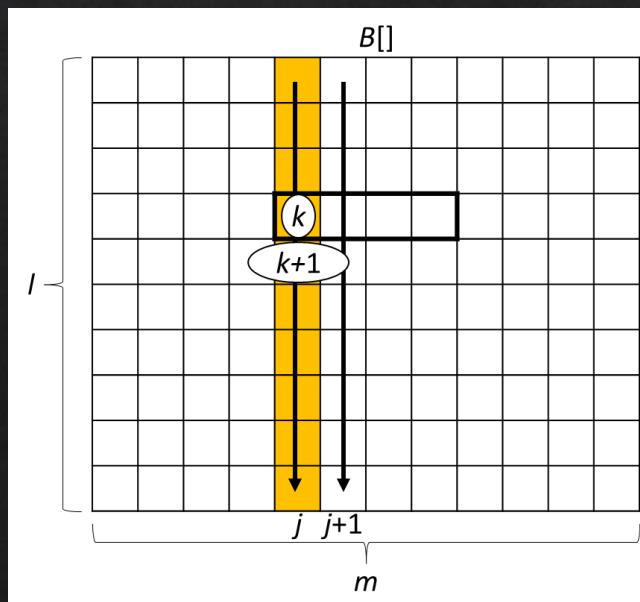
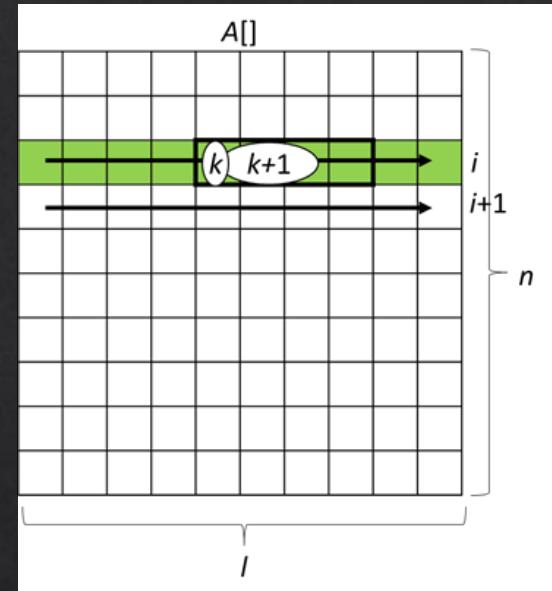
Кэш дүүрсэн бол аль өгөгдлийг зөөх вэ?

- ❖ Кэш нь дотроо хэд хэдэн Cache Line-тэй
- ❖ Кэш харгалзааны (mapping) стратеги нь үндсэн санах ойн багц бичилтийн хуулбарыг кэшд хадгалах байрлалыг тодорхойдог
- ❖ **Direct-Mapped Cache:** Үндсэн санах ойгоос блок бүрийг зөвхөн нэг кэш шугамд хадгалах боломжтой (**Cache miss** ихтэй)
- ❖ **n -way Set Associative Cache:** Үндсэн санах ойгоос блок бүрийг боломжит n кэш шугамуудын нэгэнд хадгалах боломжтой (илүү төвөгтэй, **Cache hit** сайн)
- ❖ **Least Recently Used (LRU):** Боломжит байршилуудаас Temporal Locality-д суурилж сонгодог түгээмэл зарчим.

Кэш хандалтын оновчол

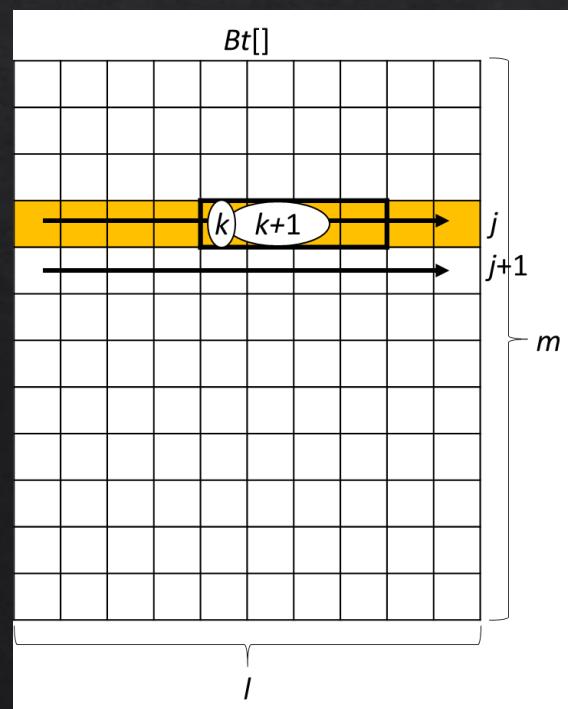
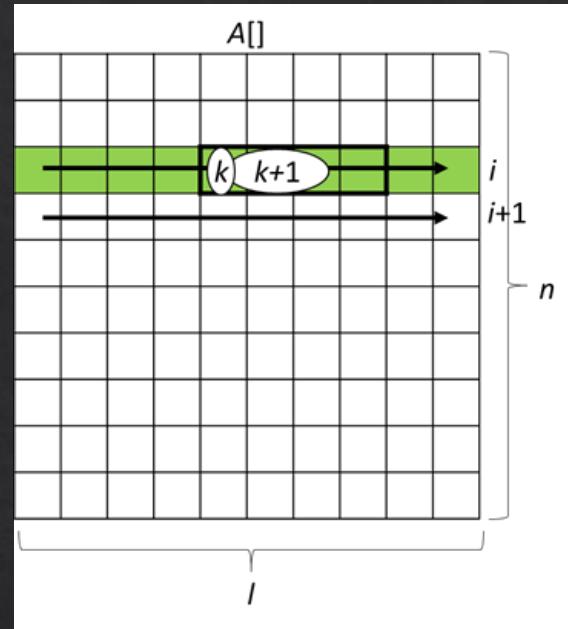
```
//Naive Matrix Multiplication
for (int i = 0; i<n; i++) {
    for (int j = 0; j < m; j++) {
        float accum = 0;
        for (k = 0; k < l; k++)
            accum += A[i*l+k]*B[k*m+j];
        C[i*m+j] = accum;
    }
}
```

- ❖ Матрицийн үржвэр: $A_{n \times l} B_{l \times m} = C_{n \times m}$
- ❖ row-major эрэмбэтэй шугаман массивуудаар хадгалсан
- ❖ A -д хандах хэв : $(i, k) \Rightarrow (i, k+1)$
- ❖ B -ийн хандагдах хэв : $(k, j) \Rightarrow (k+1, j)$
 - ❖ үндсэн санах ойд эзлэх зай $l \times \text{sizeof(float)}$ \Rightarrow кэшийн нэг шугаманд хадгалагдаагүй
 - ❖ Кэш шугам L1-кэшээс устгагдсан байх \Rightarrow том l -ийн хувьд hit-rate бага



Кэш хандалтын оновчол

```
//Transpose-and-Multiply
for (k=0; k<l; k++)
    for (j = 0; j<m; j++)
        Bt[i*l+k] = B[k*n+j];
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        float accum = 0;
        for (k=0; k<l; k++)
            accum += A[i*l+k] * B[j*l+k];
        C[i*m + j] = accum;
    }
```



❖ Transpose-and-Multiply:

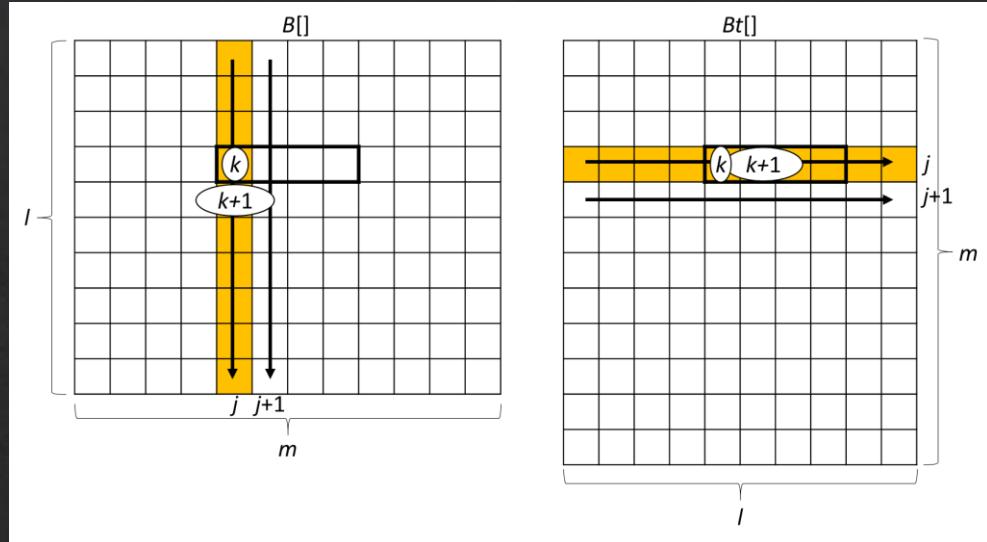
$$1. \quad Bt_{m \times l} = (B_{l \times m})^T$$

$$2. \quad A_{n \times l} Bt_{m \times l} = C_{n \times m}$$

❖ A -д хандах хэв : $(i,k) \Rightarrow (i,k+1)$

❖ B -ийн хандагдах хэв : $(j,k) \Rightarrow (j,k+1)$

Кэш хандалтын оновчол



i7-6800K: $m = n = l = 2^{13}$

```
#elapsed time (naive_mult) : 5559.5s
#elapsed time (transpose) : 0.8s
#elapsed time (transpose_mult) : 497.9s
```

Speedup:
11.1

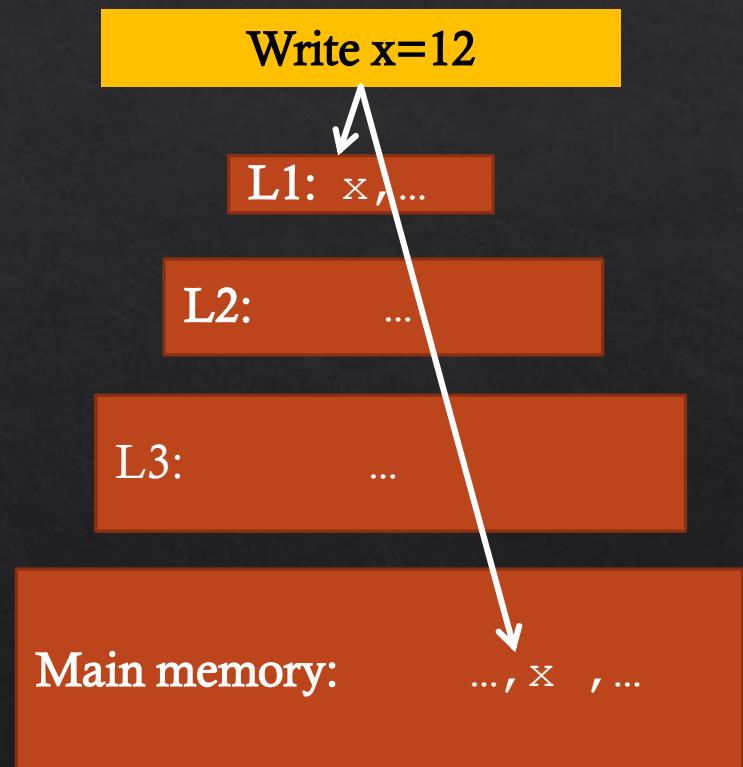
i7-6800K: $m = n = 2^{13}, l = 2^8$

```
#elapsed time (naive_mult) : 28.1s
#elapsed time (transpose) : 0.01s
#elapsed time (transpose_mult) : 12.9s
```

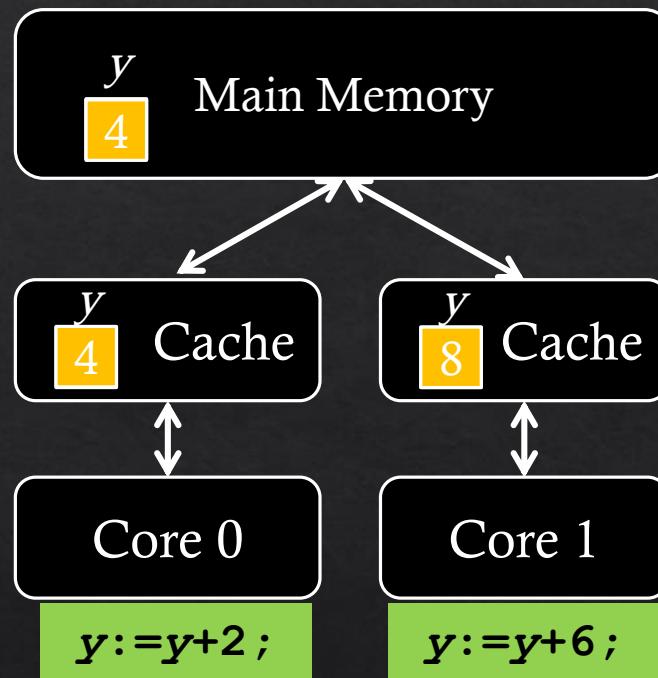
Speedup:
2.2

Кэш бичилтийн зарчмууд

- ❖ CPU нь өгөгдлийг кэш рүү бичих үед кэш дэх утга нь үндсэн санах ойн утгатай **нийцэхгүй** (inconsistent) байх боломжтой.
- ❖ Write-through
 - ❖ Кэшэд бичилт хийх үедээ үндсэн санах ойн утгийг давхар шинэчлэх замаар зохицуулдаг
- ❖ Write-back
 - ❖ Кэшүүд нь кэш дэх өгөгдлөө *бохир* гэж тэмдэглэнэ
 - ❖ Кэшийн бохир шугамыг буцаах үед л хадгалсан өгөгдлийг үндсэн санах ойд бичнэ.



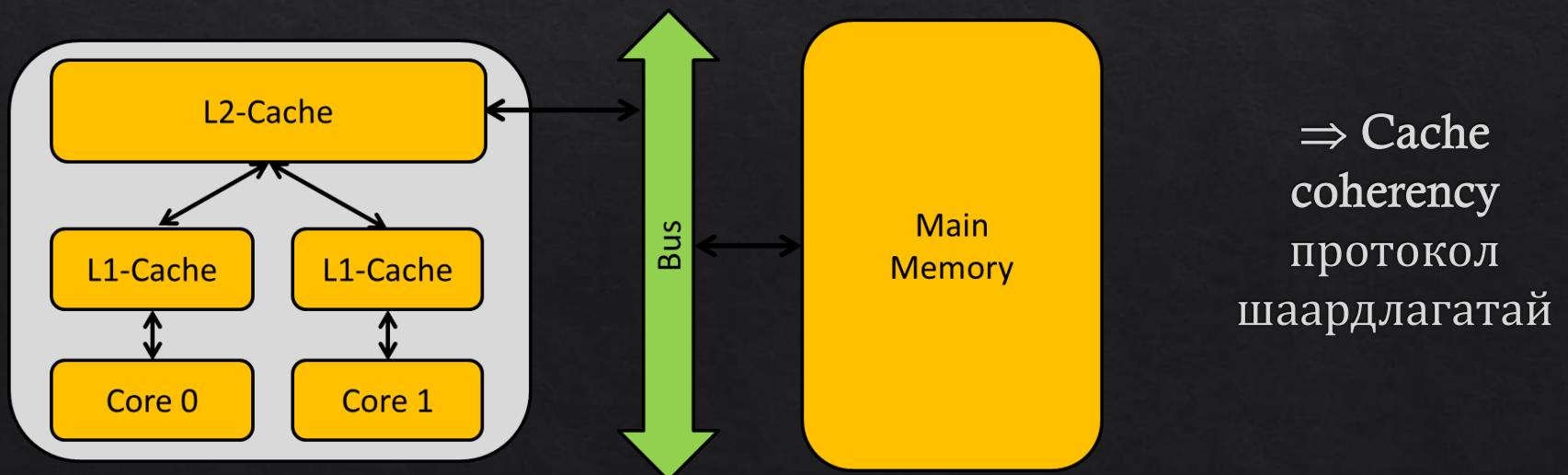
The Cache Coherence Problem – Жишээ



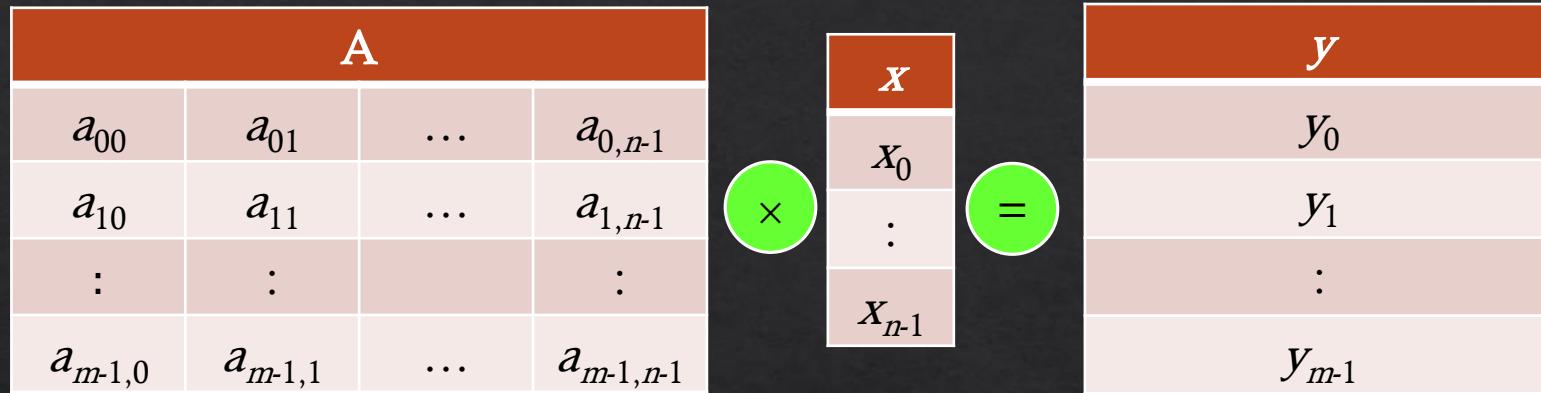
Cache inconsistency: хоёр кэш нь ижил хувьсагчид өөр өөр утгыг хадгалж байна.

Кэш Coherence

- ❖ Одоогийн зэрэгцээ цөмт CPU нь ихэвчлэн кэшийн олон түвшинтэй
 - ❖ цөм бүр нь хувийн (жижиг, хурдан) доод түвшний кэштэй байдаг
 - ❖ бүх цөм нь (том, удаан) илүү өндөр түвшний дундын кэштэй
- ❖ Дундын өгөгдөл хэд хэдэн хувь байх боломжтой
 - ❖ нэг нь Цөм 0-ийн L1-кэшд, нөгөө нь Core 1-ийн L1-кэш-д хадгалагдана
- ❖ Cache Inconsistency
 - ❖ Цөм 0 өөрийн кэш шугам руу бичвэл зөвхөн түүний L1-кэшийн утга шинэчлэгдэх боловч Core 1-ийн L1-кэшийн утга хуучнаараа байна



Матриц-Вектор үржүүлэх



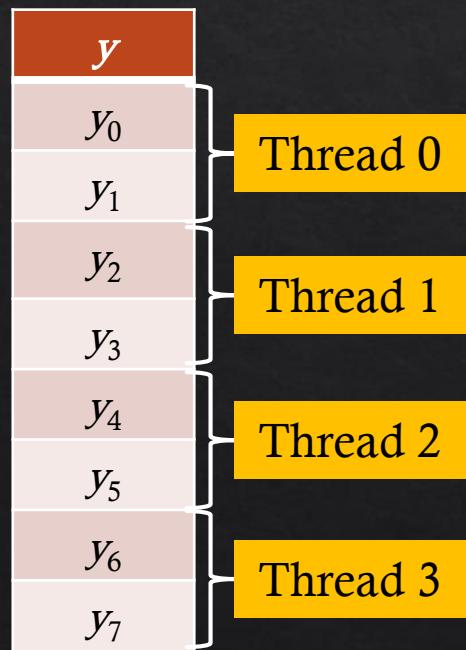
Thread	у-ын компонентууд
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

$y[i]$ -ийн хувьд ажиллах thread

```
y[i] = 0.0;  
for (j=0; j<n; j++)  
    y[i] += A[i][j] * x[j];
```

False Sharing – Cache Line Ping-Pong

```
#pragma omp parallel for schedule(static,2)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (int j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```



- ❖ Кэш coherence нь “*Cache-line level*”-д ажилладаг
- ❖ Жнь, $m = 8$ үед бүх у **нэг кэш шугамд** хадгалагдана
- ❖ False Sharing: у бичих бүрт бусад процессоруудын кэш дэх мөрийг хүчингүй болгодог. у энэ өөрлөлтүүдийн ихэнх нь thread-үүдийг үндсэн санах ой руу хандахыг шаарддаг.

Баярлалаа.