Računarske vežbe
Letnji semestar 2023/24.

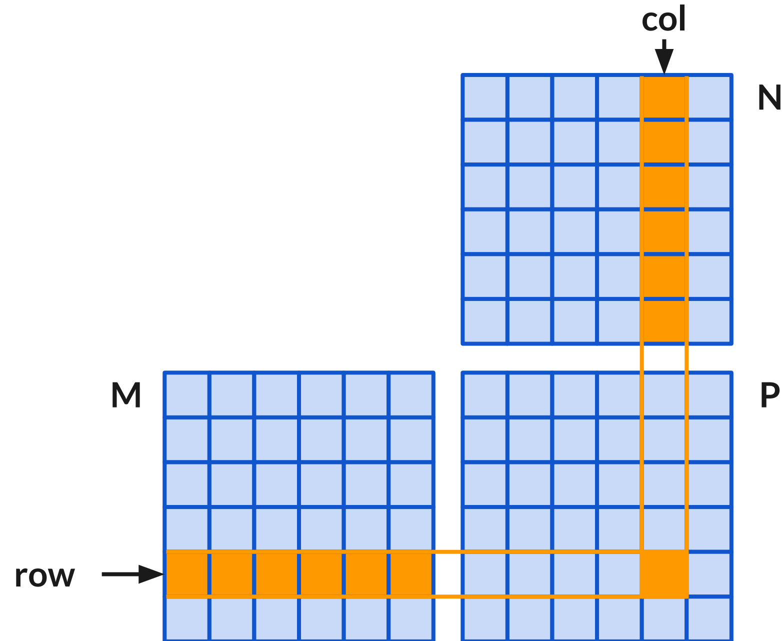
UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA PRIMENJENE RAČUNARSKE NAUKE

CUDA programiranje 2

Sadržaj

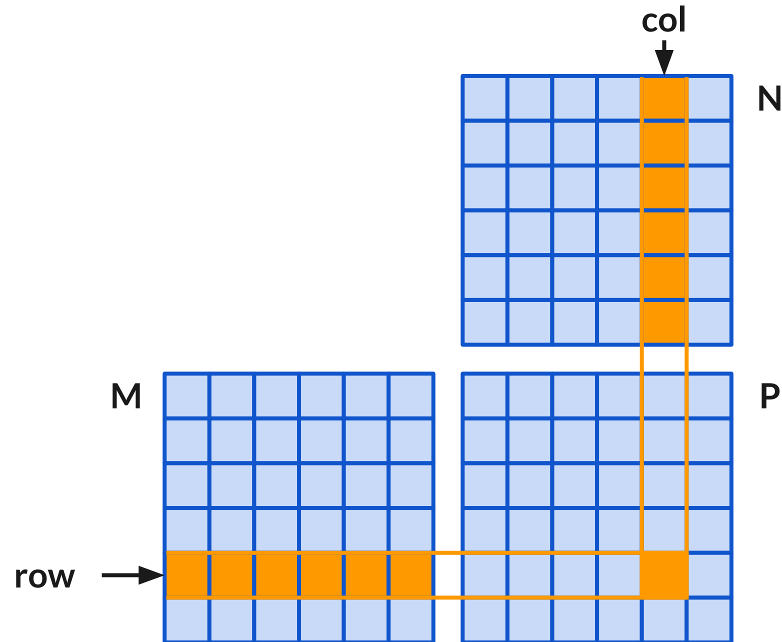
- Osnovni algoritam za množenje matrica.
- Tipovi memorije.
- Algoritam za množenje matrica sa optimizovanim pristupom memoriji.

Množenje kvadratnih matrica




```
P[row, col] = ?
```

Množenje kvadratnih matrica



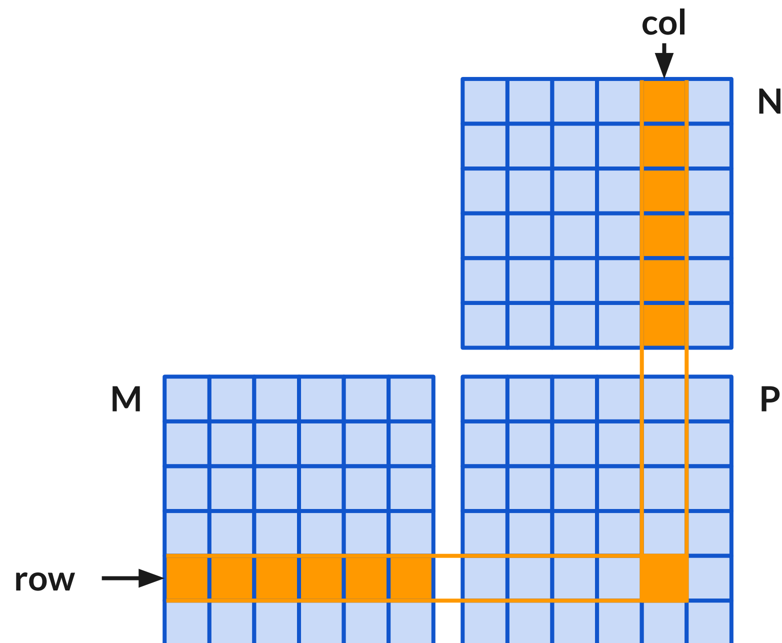

```
P[row, col] = ?
```

```
for (int i = 0; i < WIDTH; i++) {
    P[row][col] += M[row][i] * N[i][col]
}
```

Za linearizovane matrice:

`P[row][col] → ?`

Množenje kvadratnih matrica



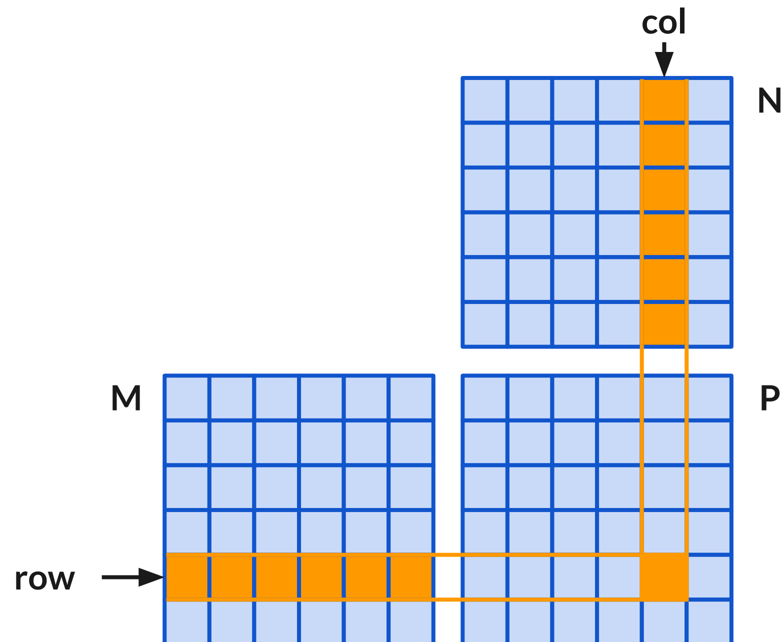

```
P[row, col] = ?
```

```
for (int i = 0; i < WIDTH; i++) {
    P[row][col] += M[row][i] * N[i][col]
}
```

Za linearizovane matrice:

```
P[row][col] → P[row * WIDTH + col]
M[row][i] → ?
```

Množenje kvadratnih matrica



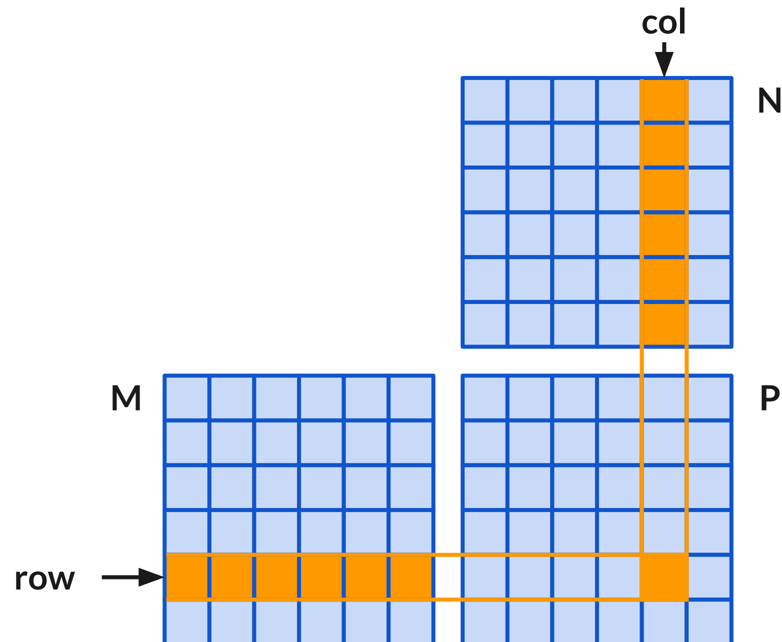

```
P[row, col] = ?
```

```
for (int i = 0; i < WIDTH; i++) {
    P[row][col] += M[row][i] * N[i][col]
}
```

Za linearizovane matrice:

```
P[row][col] → P[row * WIDTH + col]
M[row][i] → M[row * WIDTH + i]
N[i][col] → ?
```

Množenje kvadratnih matrica




```
P[row, col] = ?
```

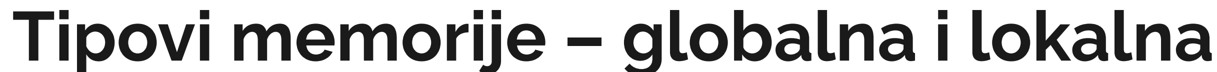
```
for (int i = 0; i < WIDTH; i++) {
     P[row][col] += M[row][i] * N[i][col]
}
```

Za linearizovane matrice:

```
P[row][col] → P[row * WIDTH + col]
M[row][i] → M[row * WIDTH + i]
N[i][col] → N[i * WIDTH +col]
```

Zadatak 1 – osnovni algoritam za množenje nekvadratnih matrica

- Prvi zadatak u *Google Colab* svesci.



Tipovi memorije – globalna i lokalna

- Globalna:
  - vidljiva svim nitima svih blokova,
  - dostupna domaćinu — za kopiranje memorije (*cudaMemcpy*), za slanje parametara kernelu,
  - spora u odnosu na ostale vidove memorije,
  - planiranim pristupom može se optimizovati njeno korišćenje.
- Lokalna
  - pojedine promenljive deklarisane unutar kernela, svaka nit ima sopstvenu kopiju (sledeći slajd sadrži detalje o tome koje promenljive deklarisane u kernelu se smeštaju u lokalnu memoriju),
  - zapravo, izolovan deo globalne memorije; skoro podjednako sporo.

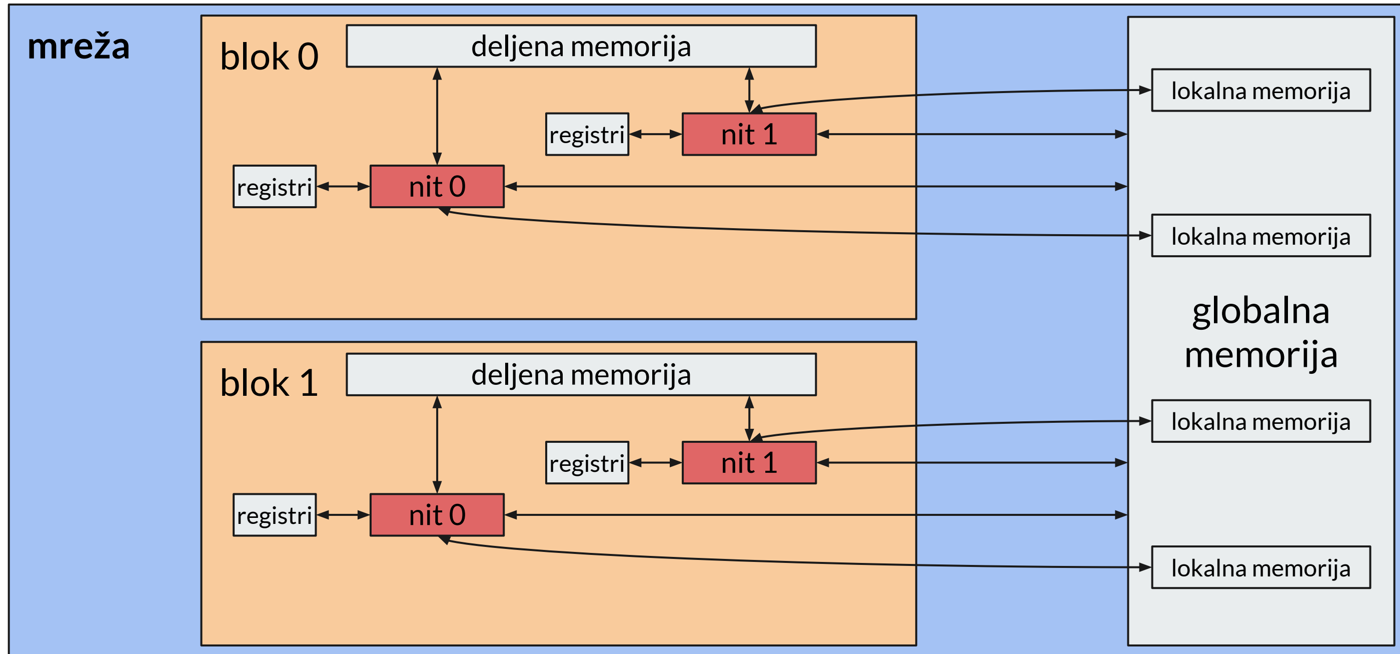
Tipovi memorije – registarska

- Ima efekat lokalne — svaka nit ima svoju privatnu.
- Najbrža, ali je ima najmanje.
- Skalarne promenljive i mali nizovi čija je veličina poznata u vreme kompajliranja **uglavnom** se smeštaju u registarsku memoriju ukoliko su deklarisani unutar kernela.
- **Uglavnom** — ne uvek; ako ih bude previše, preliće se u lokalnu memoriju.
- Nizovi čija veličina nije poznata pri kompajliranju smeštaju se u **lokalnu memoriju**.

Tipovi memorije – deljena

- Postoji na nivou **bloka niti**.
- Sve niti istog bloka vide istu deljenu memoriju, ali **nemaju pristup** deljenoj memoriji drugih blokova.
- **Brža** od globalne.
- Deklarisana unutar kernela pomoću oznake `__shared__`.

Tipovi memorije



Tipovi memorije – primer

```
__global__ void MatrixMulKernel(float* globArr, int globVarLocArrSize) {

    __shared__ float shArr[SH_ARR_SIZE];

    int regTx = threadIdx.x;
    regArr[REG_ARR_SIZE];

    int *locArr = (int*)malloc(globVarLocArrSize * sizeof(int));
}
```

Tipovi memorije – primer

```
__global__ void MatrixMulKernel(float* globArr, int globVarLocArrSize) {

    __shared__ float shArr[SH_ARR_SIZE];

    int regTx = threadIdx.x;
    regArr[REG_ARR_SIZE];

    int *locArr = (int*)malloc(globVarLocArrSize * sizeof(int));
}
```

Globalna memorija – parametri prosleđeni od strane domaćina.

Tipovi memorije – primer

```
__global__ void MatrixMulKernel(float* globArr, int globVarLocArrSize) {

    __shared__ float shArr[SH_ARR_SIZE];

    int regTx = threadIdx.x;
    regArr[REG_ARR_SIZE];

    int *locArr = (int*)malloc(globVarLocArrSize * sizeof(int));
}
```

Deljena memorija – označena sa `__shared__`.

Tipovi memorije – primer

```
__global__ void MatrixMulKernel(float* globArr, int globVarLocArrSize) {

    __shared__ float shArr[SH_ARR_SIZE];

    int regTx = threadIdx.x;
    regArr[REG_ARR_SIZE];

    int *locArr = (int*)malloc(globVarLocArrSize * sizeof(int));
}
```

Registarska memorija – skalarna promenljiva deklarisana unutar kernela. Da ima previše registarskih promenljivih, neke bi bile prebačene u lokalnu memoriju.

Tipovi memorije – primer

```
__global__ void MatrixMulKernel(float* globArr, int globVarLocArrSize) {

    __shared__ float shArr[SH_ARR_SIZE];

    int regTx = threadIdx.x;
    regArr[REG_ARR_SIZE];

    int *locArr = (int*)malloc(globVarLocArrSize * sizeof(int));
}
```

Registarska memorija – statički niz deklarisan unutar kernela. **`REG_ARR_SIZE`** je konstanta. Da je ovaj niz prevelik za registarsku memoriju, bio bi prebačen u lokalnu.

Tipovi memorije – primer

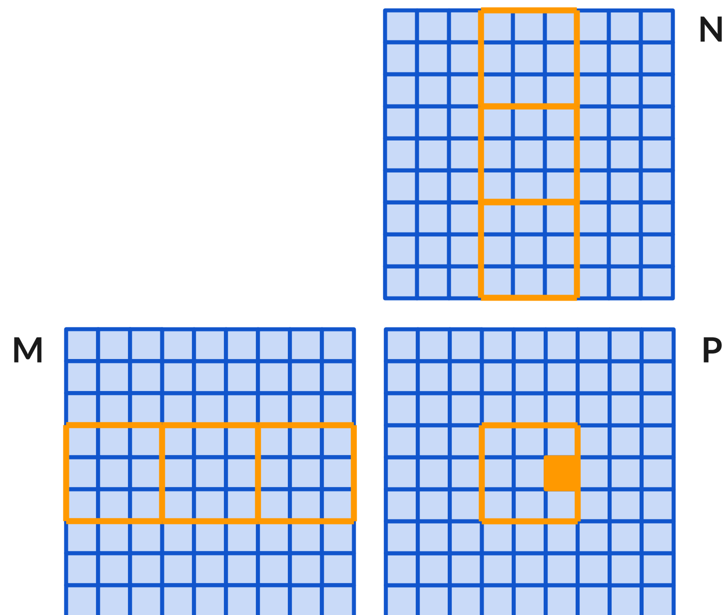
```
__global__ void MatrixMulKernel(float* globArr, int globVarLocArrSize) {

    __shared__ float shArr[SH_ARR_SIZE];

    int regTx = threadIdx.x;
    regArr[REG_ARR_SIZE];

    int *locArr = (int*)malloc(globVarLocArrSize * sizeof(int));
}
```

Lokalna memorija – dinamički alociran niz.

Množenje matrica popločavanjem

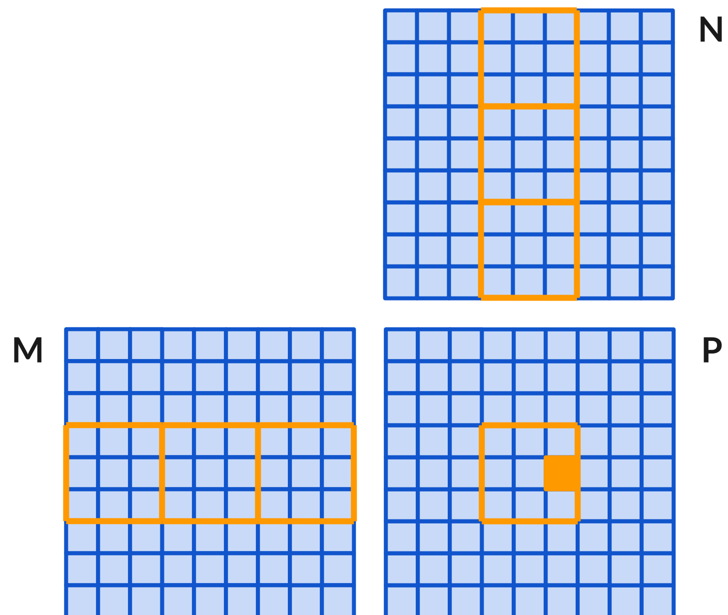
- Engl. *Tiled matrix multiplication.*
- Ideja — smanjiti broj pristupa globalnoj memoriji saradnjom niti unutar blokova i korišćenjem deljene memorije.
- Račun se odvija kroz faze gde se iz globalne u deljenu memoriju učitava po jedna pločica (engl. *tile*) matrica N i M:
    - za TILE_SIZE == BLOCK_SIZE svaka nit u bloku učitava samo 2 elementa po fazi (1 za pločicu iz matrice N, 1 za pločicu iz matrice M)

Množenje matrica popločavanjem



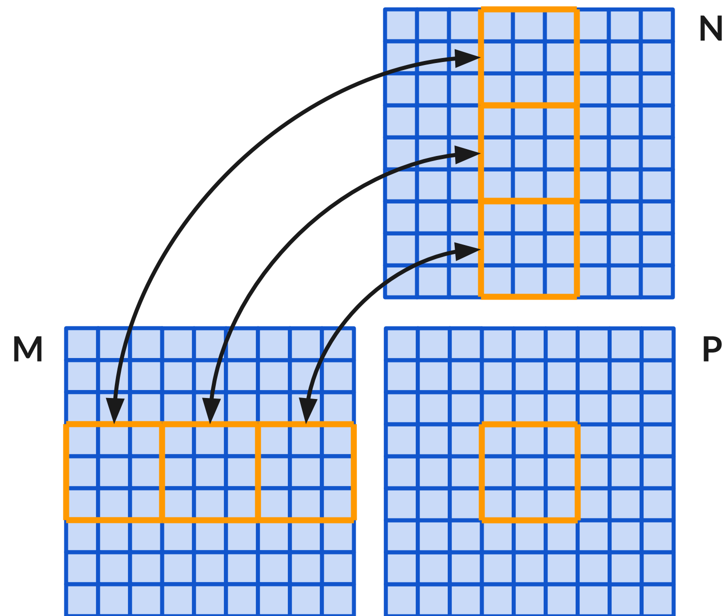
- Jedan **blok** niti računa elemente jedne **pločice** matrice P.
- Za računanje vrednosti **jedne pločice matrice P**, potrebno je iskoristiti **jednu vrstu pločica iz M** i **jednu kolonu pločica iz N**.
- **Jedna nit** računa **jedan element** izlazne matrice.

Množenje matrica popločavanjem



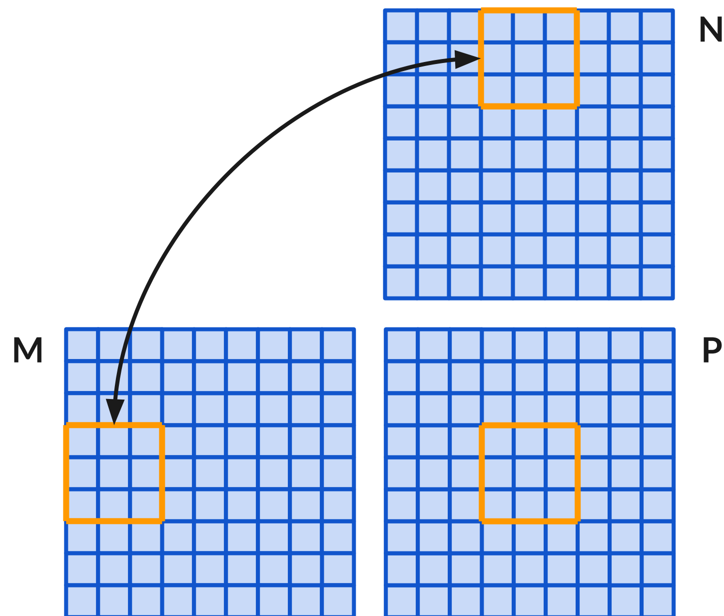
- Blokova niti ima koliko i pločica u matrici P.
- Ukupno ima onoliko niti koliko i elemenata u matrici P.
- To znači da matrica P ima identičan izgled kao *CUDA* mreža blokova i niti koja se pokreće.

Množenje matrica popločavanjem



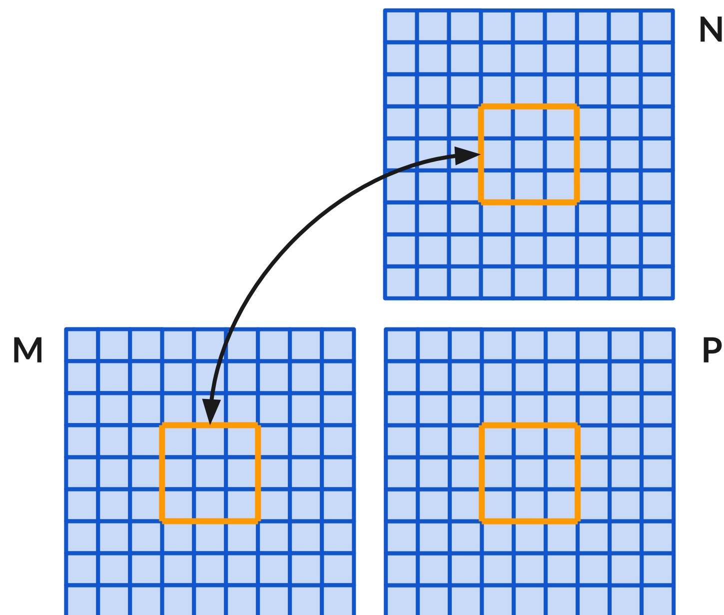
- Svaki par pločica iz M i N čini po jednu fazu u računanju pločice P.

Množenje matrica popločavanjem



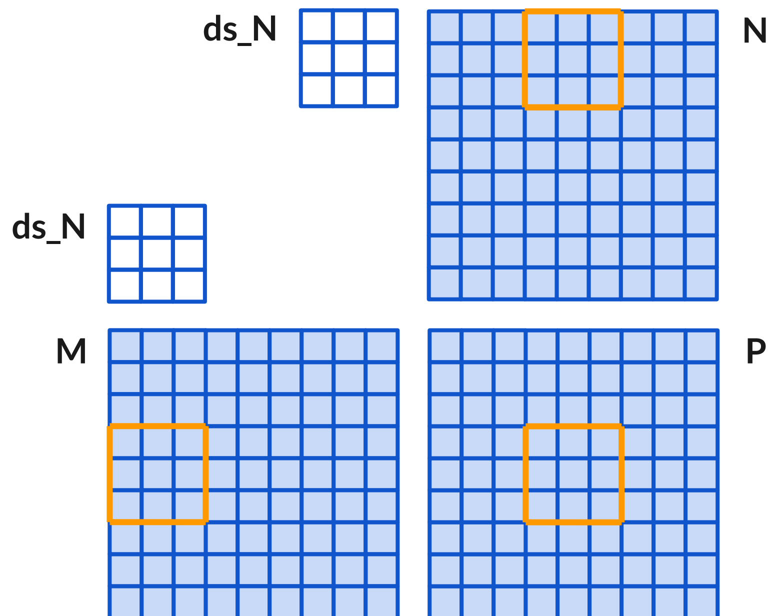
- Svaki par pločica iz M i N čini po jednu fazu u računanju pločice P.
- Prva pločica iz vrste pločica matrice M i prva pločica iz kolone pločica matrice N čine prvu fazu (fazu 0, ako brojanje kreće od 0).

Množenje matrica popločavanjem



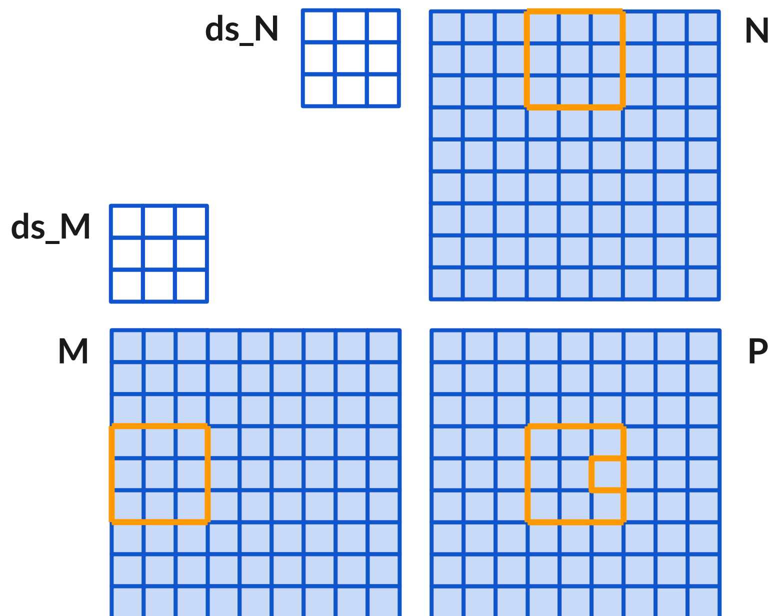
- Svaki par pločica iz M i N čini po jednu fazu u računanju pločice P.
- Prva pločica iz vrste pločica matrice M i prva pločica iz kolone pločica matrice N čine prvu fazu (fazu 0, ako brojanje kreće od 0).
- Sledeće dve čine drugu fazu (fazu 1), itd.

Množenje matrica popločavanjem



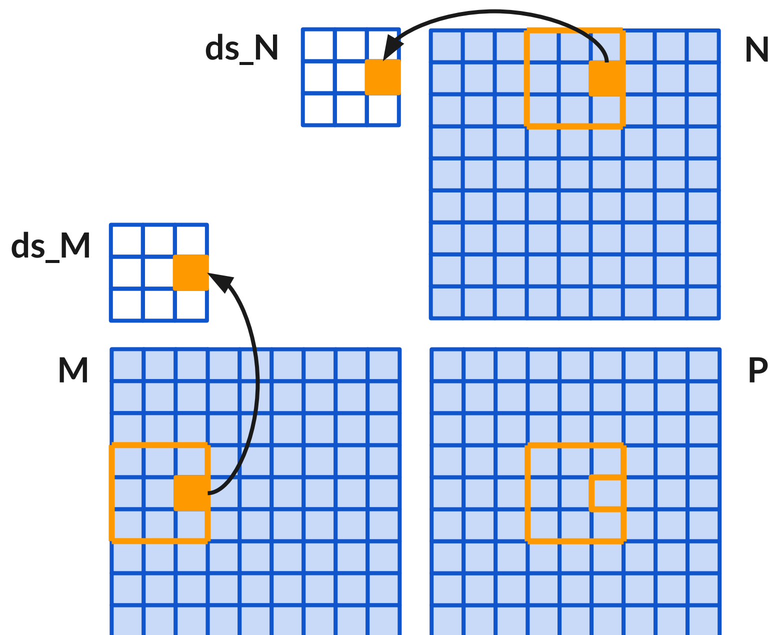
- U svakoj fazi, trenutnim pločicama iz matrica M i N pridružena je deljena memorija iste veličine kao i pločice.
- ds_M za matricu M, ds_N za matricu N.

Množenje matrica popločavanjem



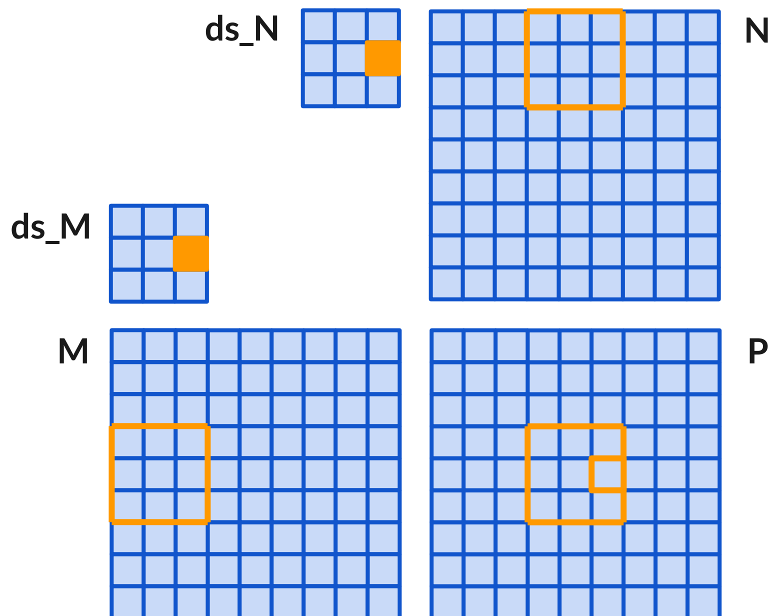
- U svakoj fazi, **nit** koja računa **odgovarajući element** matrice P: ...

Množenje matrica popločavanjem



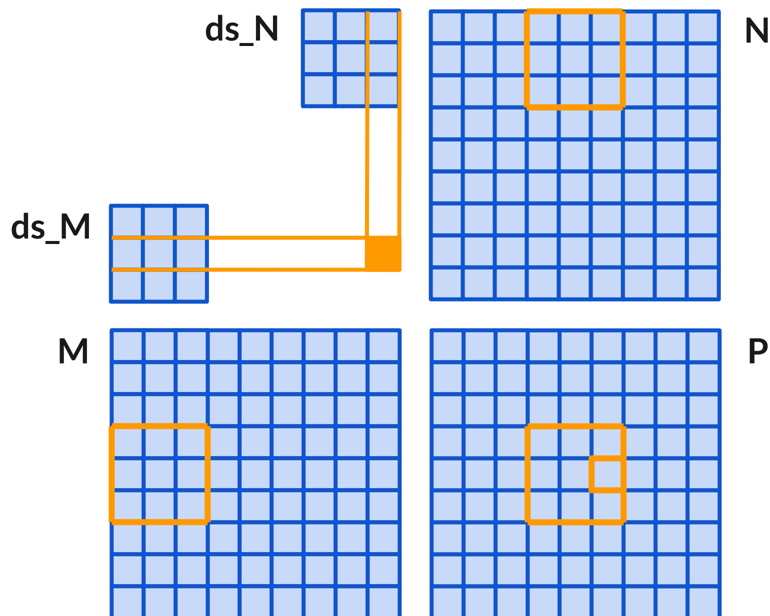
- U svakoj fazi, **nit** koja računa **odgovarajući element** matrice P:
  a. učitava odgovarajuće elemente iz M i N na odgovarajuća mesta u ds_M i ds_N...

Množenje matrica popločavanjem



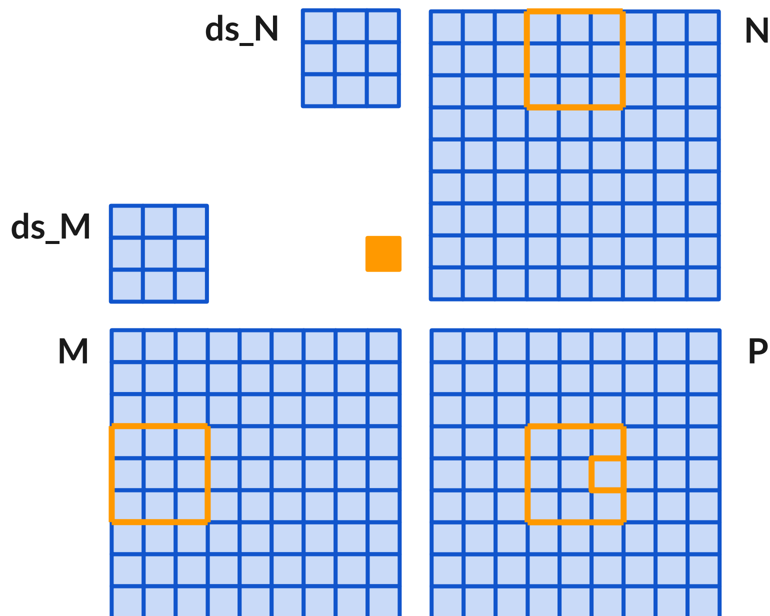
- U svakoj fazi, **nit** koja računa **odgovarajući element** matrice P:
  a. učitava odgovarajuće elemente iz M i N na odgovarajuća mesta u ds_M i ds_N,
  b. čeka da ostale niti iz bloka učitaju svoje elemente u ds_M i ds_N...

Množenje matrica popločavanjem



- U svakoj fazi, **nit** koja računa **odgovarajući element** matrice P:
  a. učitava odgovarajuće element iz M i N na odgovarajuća mesta u ds_M i ds_N,
  b. čeka da ostale niti iz bloka učitaju svoje elemente u ds_M i ds_N...
  c. množi odgovarajuću vrstu iz ds_M i kolonu iz ds_N, i smešta rezultat u privremenu promenljivu, i...

Množenje matrica popločavanjem



- U svakoj fazi, **nit** koja računa **odgovarajući element** matrice P:
    a. učitava odgovarajuće element iz M i N na odgovarajuća mesta u ds_M i ds_N,
    b. čeka da ostale niti iz bloka učitaju svoje elemente u ds_M i ds_N...
    c. množi odgovarajuću vrstu iz ds_M i kolonu iz ds_N, i smešta rezultat u privremenu promenljivu, i
    d. čeka da ostale niti odrade svoje množenje.

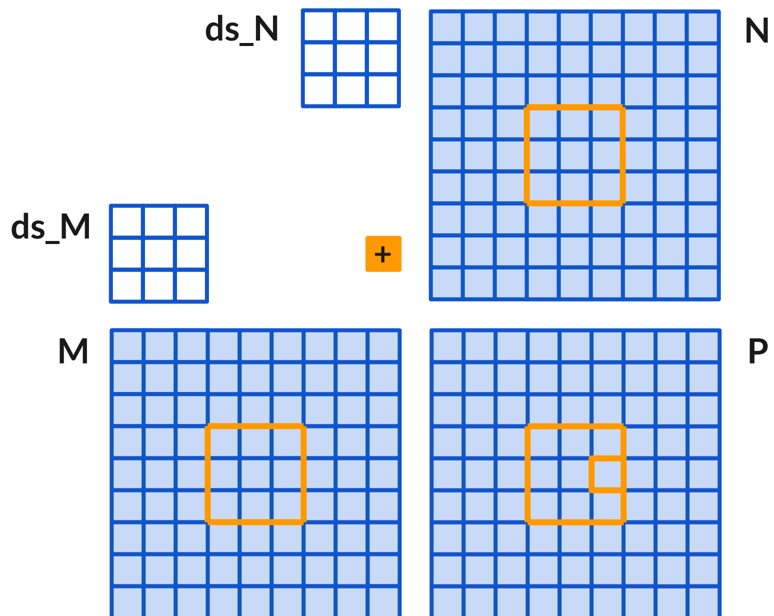
Zašto u svakoj fazi postoje čekanja između niti?

- Čekanja predstavljaju sinhronizaciju.
- Sinhronizacija u koraku *b* na prethodnim slajdovima je uvedena jer nit u koraku *c* množi ne samo elemente koje je sama učitala u ds_M i ds_N, već i elemente koje su tamo učitale druge niti; nakon sinhronizacije, garantuje se da su sve niti učitale svoje elemente.
- Sinhronizacija u koraku *d* na prethodnim slajdovima je uvedena jer se iste deljene memorije (ds_M i ds_N) koriste u svakoj fazi, pa nit mora da sačeka da sve niti koje koriste njene elemente iz ds_M i ds_N završe množenje, pre nego što ih prepiše novim vrednostima u narednoj fazi.

Funkcija za sinhronizaciju

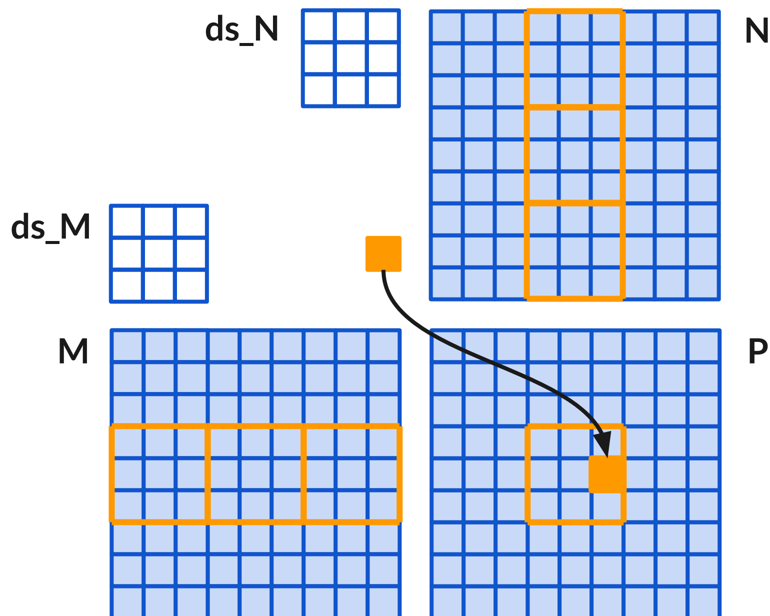
- Koristi se __*syncthreads()* funkcija koja pravi barijeru na kojoj se zaustavljaju sve niti **iz istog bloka.**
- Izvršenje se nastavlja kada sve niti iz bloka stignu do barijere.
- Pogodno kada niti proizvode rezultat koji će možda biti potreban ostalim nitima iz tog bloka.
- Da li ovo liči na nešto već viđeno u *OpenMP*-u?

Množenje matrica popločavanjem



- Naredna faza se sastoji od identičnih koraka, samo se koriste druge pločice iz M i N.
- Rezultat množenja iz druge faze se dodaje na prethodni.

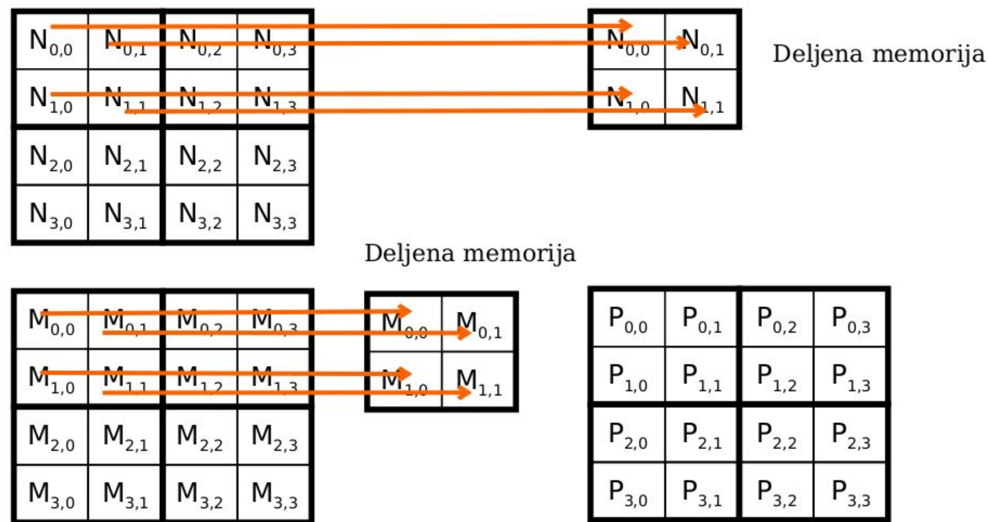
Množenje matrica popločavanjem



- Na kraju svih faza, rezultat se iz privremene promenljive upisuje u matricu P.
- Privremenom promenljivom smanjuje se broj pristupa globalnoj memoriji za upis rezultata, jer se upis u globalnu memoriju vrši samo jednom, umesto nakon svake faze.

Faza 0 — učitavanje pločica u deljenu memoriju

- Primer : Blok 0 (dims = (2, 2)), *TILE_SIZE* = 2



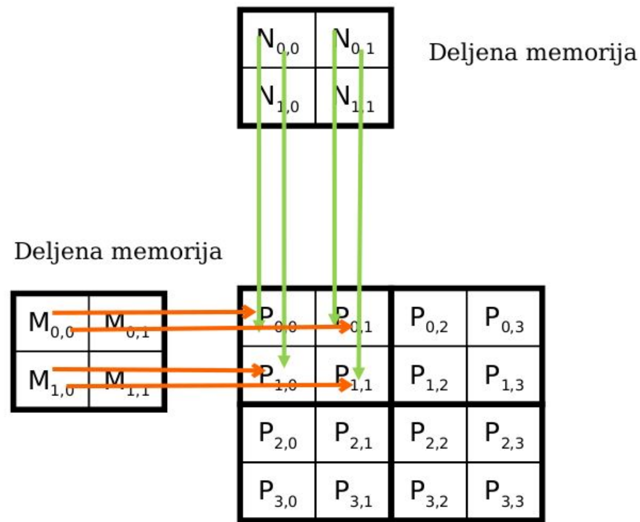
CUDA

Faza 0 — računanje parcijalnog proizvoda (iteracija 0)

- Primer : Blok 0

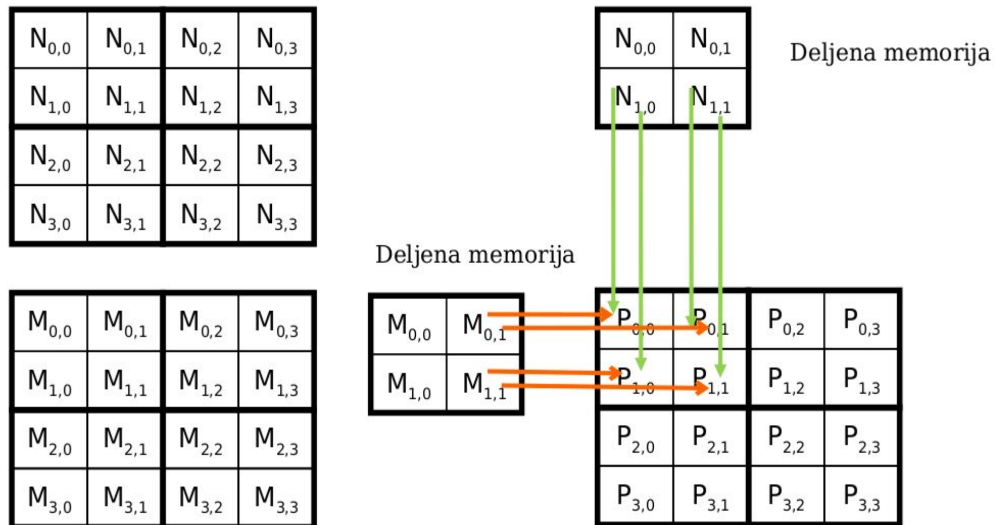
$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



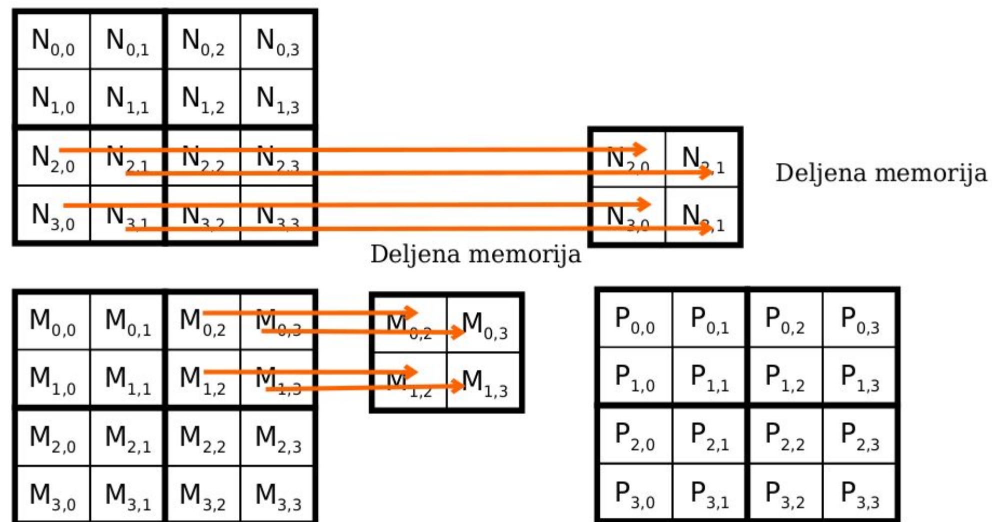
Faza 0 — računanje parcijalnog proizvoda (iteracija 1)

- Primer : Blok 0



Faza 1 — učitavanje pločica u deljenu memoriju

- Primer : Blok 0

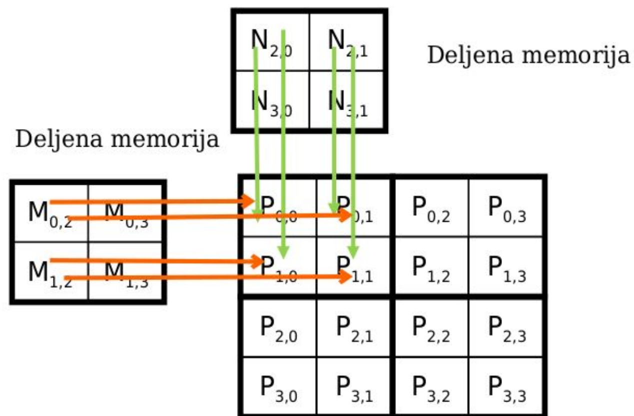


Faza 1 — računanje parcijalnog proizvoda (iteracija 0)

- Primer : Blok 0

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

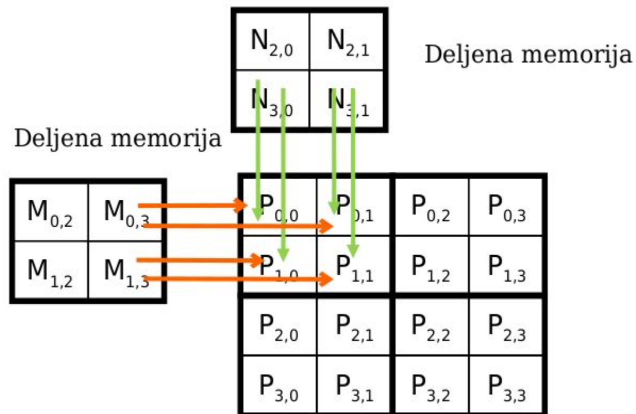


Faza 1 — računanje parcijalnog proizvoda (iteracija 1)

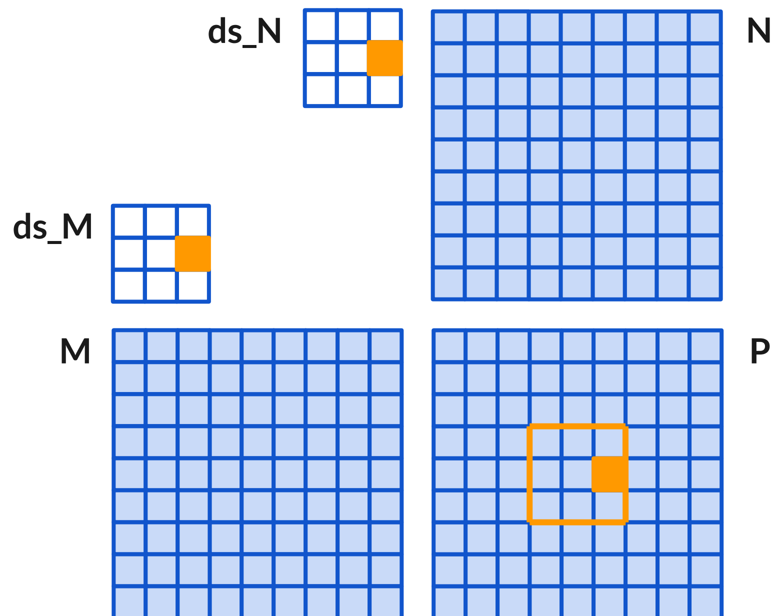
- Primer : Blok 0

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



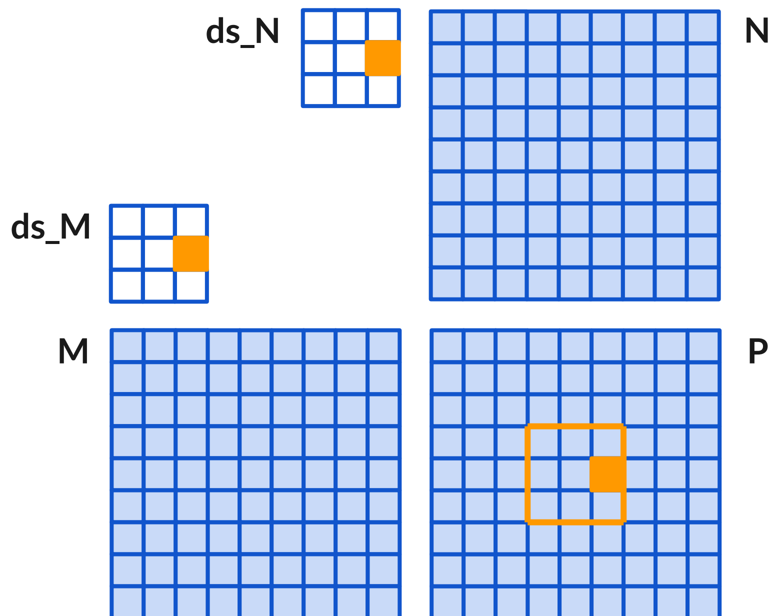
Učitavanje pločica u deljnu memoriju



Indeksiranje deljene memorije

- Nit će deljenu memoriju pri upisu svog elementa uvek indeksirati isto, nezavisno od faze.
- Indeksi elementa u deljenoj memoriji isti su kao indeksi niti unutar njenog bloka.
- Nit indeksira deljenu memoriju pomoću *threadIdx.y* i *threadIdx.x*.

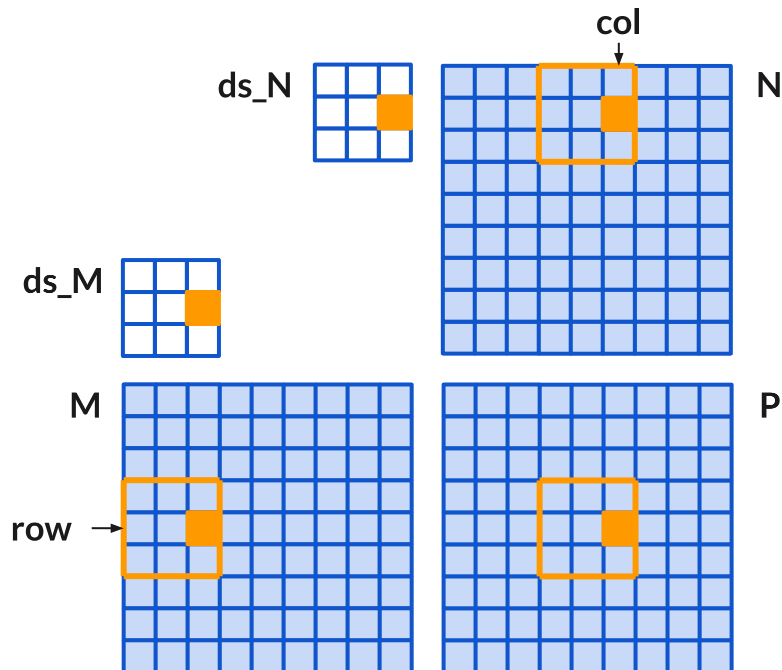
Učitavanje pločica u deljnu memoriju



Indeksiranje deljene memorije

- Na ilustraciji, nit je unutar bloka na indeksima:
    - *threadIdx.y = 1*
    - *threadIdx.x = 2*
- Indeksi upisanog elementa u ds_M i ds_N su identični.

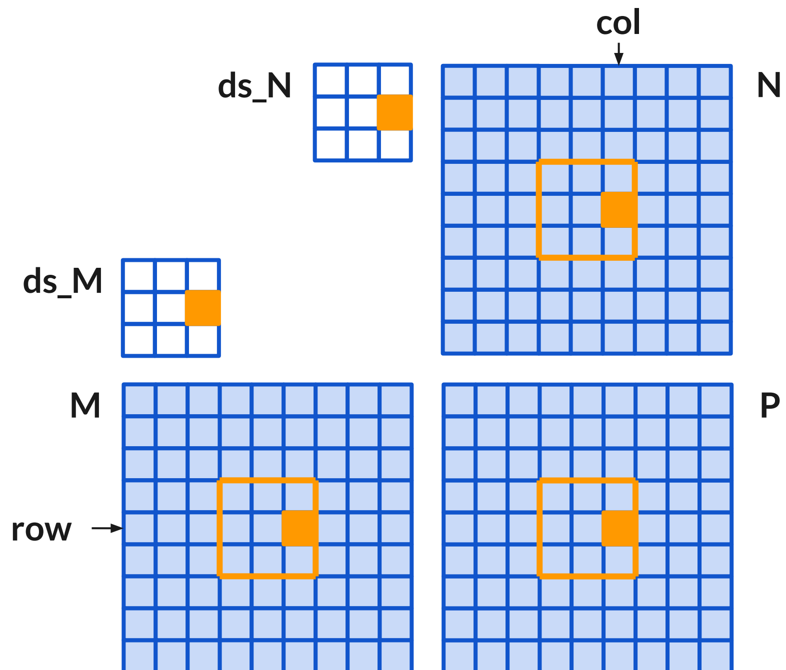
Učitavanje pločica u deljnu memoriju



Indeksiranje M i N

- Vrednosti *row* i *col* se dobijaju kao:
  - *row = blockIdx.y * blockDim.y + threadIdx.y*
  - *col = blockIdx.x * blockDim.x + threadIdx.x*
- U fazi 0, matrice se indeksiraju na sledeći način:
  - *M[row][tx]*
  - *N[ty][col]*

Učitavanje pločica u deljnu memoriju



Indeksiranje M i N

- Vrednosti *row* i *col* se dobijaju kao:
  - *row = blockIdx.y * blockDim.y + threadIdx.y*
  - *col = blockIdx.x * blockDim.x + threadIdx.x*
- U fazi 1, matrice se indeksiraju na sledeći način:
  - *M[row][1 * TILE_SIZE + tx]*
  - *N[1 * TILE_SIZE + ty][col],*
  - gde je *TILE_SIZE* dimenzija pločice.

1D indeksiranje za pločice iz M i N

- Nizovi M i N su dinamički alocirani, koristi se 1D indeksiranje.
- Ukupno pločica: WIDTH/TILE_SIZE
- Globalni indeksi kolone u matrici M i vrste u matrici N zavise od faze, odnosno rednog broja pločice.
- p – redni broj pločice

```
M[row][p*TILE_SIZE+tx] → M[row*WIDTH+p*TILE_SIZE+tx]

N[p*TILE_SIZE+ty][col] → N[(p*TILE_SIZE+ty)*WIDTH+col]
```

MatrixMultiplicationTiled kernel

CUDA

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    for (int p = 0; p < Width/TILE_WIDTH; ++p) {
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH + tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH + ty)*WIDTH + Col];
        __syncthreads( );

        for(int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads( );
    }
    P[Row*Width+Col] = Pvalue
}
```

inicijalizacija

CUDA

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
     __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
     __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

     int bx = blockIdx.x, by = blockIdx.y;
     int tx = threadIdx.x, ty = threadIdx.y;

     int Row = by * blockDim.y + ty;
     int Col = bx * blockDim.x + tx;
     float Pvalue = 0;

     for (int p = 0; p < Width/TILE_WIDTH; ++p) {
          ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH + tx];
          ds_N[ty][tx] = N[(p*TILE_WIDTH + ty)*WIDTH + Col];
          __syncthreads( );

          for(int i = 0; i < TILE_WIDTH; ++i)
               Pvalue += ds_M[ty][i] * ds_N[i][tx];
          __syncthreads( );
     }
     P[Row*Width+Col] = Pvalue
}
```

učitavanje elemenata matrica M i N u deljenu memoriju

CUDA

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
     __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
     __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

     int bx = blockIdx.x, by = blockIdx.y;
     int tx = threadIdx.x, ty = threadIdx.y;

     int Row = by * blockDim.y + ty;
     int Col = bx * blockDim.x + tx;
     float Pvalue = 0;

     for (int p = 0; p < Width/TILE_WIDTH; ++p) {
          ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH + tx];
          ds_N[ty][tx] = N[(p*TILE_WIDTH + ty)*WIDTH + Col];
          __syncthreads( );

          for(int i = 0; i < TILE_WIDTH; ++i)
               Pvalue += ds_M[ty][i] * ds_N[i][tx];
          __syncthreads( );
     }
     P[Row*Width+Col] = Pvalue
}
```

računanje parcijalnog proizvoda i akumuliranje rezultata