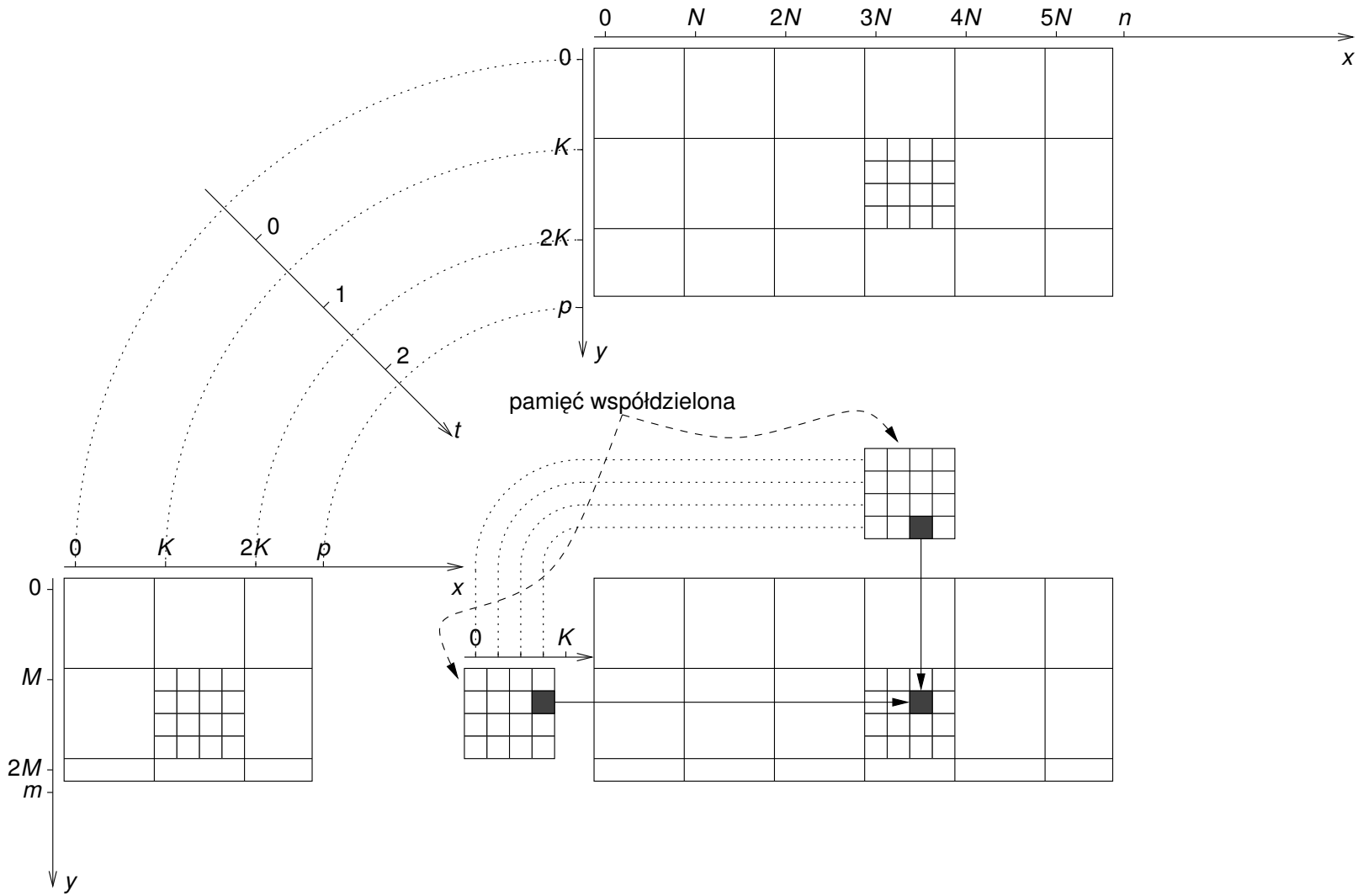
Mnożenie macierzy raz jeszcze – technika „kafelkowa”

- Przetwarzanie danych porcjami oznacza, że na „kafelki” musimy podzielić nie tylko macierz $\boldsymbol{C}$, ale także $\boldsymbol{A}$ i $\boldsymbol{B}$.
- Ze względu na łatwość zapisu i symetrię algorytmu przyjmiemy, kwadratowe „kafelki” $K \times K$ ($K = M = N$).
- Zmieniamy „ziarnistość” jądra (ale nie wątku!): algorytm opisujemy i realizujemy blokowo – dla każdego „kafelka” $\boldsymbol{C}$.
- Obliczenie elementów „kafelka” $\boldsymbol{C}$ przebiega etapami, podczas których skokowo poruszamy się po „kafelkach” $\boldsymbol{A}$ (w prawo blokowego wiersza) i $\boldsymbol{B}$ (w dół blokowej kolumny):

Mnożenie macierzy techniką „kafelkową” – algorytm

1: **for all** „kafelek” macierzy ***C*** **do**
2: wyzeruj akumulator każdego elementu „kafelka” ***C***
3: **for all** „kafelek” wiersza ***A*** i kolumny ***B*** **do**
4: pobierz kolejne kafelki ***A*** i ***B*** do pamięci współdzielonej
5: poczekaj, aż oba „kafelki” zostaną całkowicie pobrane
6: dodaj do akumulatora każdego elementu „kafelka” ***C*** cząstkowe iloczyny skalarne „kafelków” ***A*** i ***B***
7: poczekaj, aż wszystkie cząstkowe iloczyny skalarne pobranych „kafelków” ***A*** i ***B*** zostaną dodane
8: **end for**
9: zapisz akumulator każdego elementu „kafelka” ***C***
10: **end for**

Mnożenie macierzy techniką „kafelkową” – obliczenia



Zyski z mnożenia macierzy techniką „kafelkową”

- Dzięki identycznym i kwadratowym „kafelkom” każdy wątek ładuje do pamięci współdzielonej po jednym elemencie macierzy $\boldsymbol{A}$ i $\boldsymbol{B}$, odpowiadającym mu „geograficznie”.
- Z raz pobranego elementu $\boldsymbol{A}$ ($\boldsymbol{B}$) korzysta potem K wątków odpowiadających elementom danego wiersza (kolumny) $\boldsymbol{C}$.
- Wymagana liczba odczytów danych z pamięci globalnej maleje K razy. Współczynnik pamięciowy złożoności obliczeniowej (CGMA) wynosi teraz w przybliżeniu K.
- Na obu etapach przetwarzania (kopiowanie i obliczenia) przetwarzamy tylko mały fragment danych wejściowych, uzyskując lokalność algorytmu w dostępach do pamięci.
- W każdym obiegu pętli algorytmu (dla „kafelka” $\boldsymbol{C}$) dokonujemy dwóch synchronizacji wątków na barierze (poczekaj). Ponieważ jest to możliwe tylko w ramach bloku, więc musimy utożsamić „kafelek” macierzy $\boldsymbol{C}$ z blokiem.
- Warunkowa synchronizacja - ryzyko zakleszczenia!

Jądro mnożenia macierzy w CUDA C – `matmul4.cu`

- Ponieważ będziemy badać wpływ rozmiaru „kafelka” na efektywność programu, więc pamięć współdzieloną musimy zadeklarować dynamicznie, a nie statycznie.
- Potrzebujemy pamięci na dwa „kafelki” $K \times K$.

```
1  void matmul(float *C, float *A, float *B,
2              int m, int p, int n, ...)
3  { ...
4    dim3 dimBlock(K, K), dimGrid(gridSizeX, gridSizeY);
5    size_t shSize = 2 * K * K * sizeof(float);
6    ...
7    matmul_kernel<<<dimGrid, dimBlock, shSize>>>
8                  (dev_C, dev_A, dev_B, m, p, n);
9    ...
10 }
```

- Dynamicznie można zaalokować tylko jedną tablicę, a potrzebujemy dwóch...
- Musimy pamiętać o obliczeniach dla skrajnych „kafelków”.

Jądro mnożenia macierzy w CUDA C – `matmul4.cu`

```
1  __global__
2  static void matmul_kernel(
3    float *C, float *A, float *B, int m, int p, int n)
4  {
5    extern __shared__ float sh_mem[];
6    float *As = &sh_mem[0];
7    float *Bs = &sh_mem[blockDim.x * blockDim.y];
8    int m1 = ((m+blockDim.y-1) / blockDim.y) * blockDim.y;
9    int n1 = ((n+blockDim.x-1) / blockDim.x) * blockDim.x;
10   int pp =  (p+blockDim.x-1) / blockDim.x;
11
12   for (int i = threadIdx.y + blockIdx.y*blockDim.y; i < m1;
13                              i += gridDim.y*blockDim.y) {
14   for (int j = threadIdx.x + blockIdx.x*blockDim.x; j < n1;
15                              j += gridDim.x*blockDim.x) {
16   float s = 0;
17   ...
18
19
20
```

Jądro mnożenia macierzy w CUDA C – matmul4.cu

```
1     float s = 0;
2
3     int idx = threadIdx.y*blockDim.x + threadIdx.x;
4     for (int t = 0; t < pp; t++) // pętla po "kafelkach"
5     {
6       int kx = threadIdx.x + t*blockDim.x;
7       int ky = threadIdx.y + t*blockDim.y;
8       As[idx] = (i < m && kx < p) ? A[i*p + kx] : 0;
9       Bs[idx] = (j < n && ky < p) ? B[ky*n + j] : 0;
10      __syncthreads();
11
12      for (int k = 0; k < blockDim.x; k++)
13        s += As[threadIdx.y*blockDim.y + k] *
14             Bs[k*blockDim.x + threadIdx.x];
15      __syncthreads();
16    }
17
18    if (i < m && j < n)
19      C[i*n + j] = s;
20  }}}
```

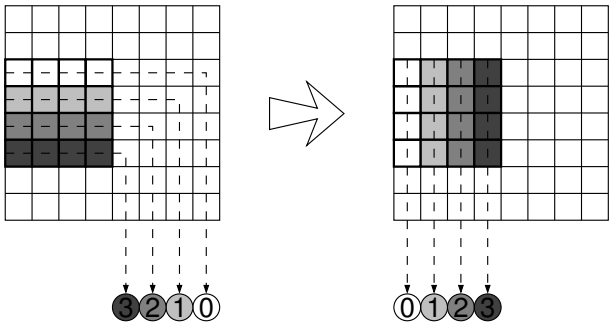
Grupowanie dostępów do pamięci globalnej

- Pamięć globalna jest realizowana jako pamięć DRAM – stan logiczny jest zapamietywany jako ładunek bardzo małego kondensatora,...
- ...tak małego, że po każdym odczycie trzeba sprawdzać, czy zanadto go nie rozładowaliśmy. Jest to tak kosztowne, że dane nie są odczytywane pojedynczo, tylko całymi liniami (z kolejnych komórek).
- Procesory GPU potrafią to wykorzystać, grupując w jedną transakcję (rozmiaru $32^{1.2,\geqslant 6.0}$, 64 lub $128^{3.5,5.2}$ B) odczyty pochodzące z jednego splotu wątków spod kolejnych (odpowiednio wyrównanych) adresów (*coalescence*).
- CUDA $\geqslant 11 \wedge C_c \geqslant 8.0 \Rightarrow$ można pozostawiać dane w L2.

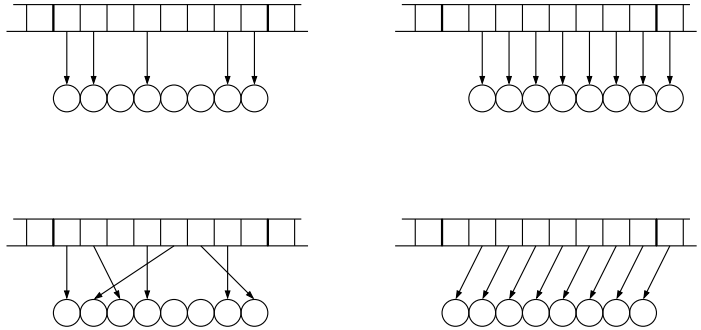
Grupowanie dostępów do pamięci globalnej...

...następuje, gdy wątki splotu (o kolejnych identyfikatorach)$^{<6.0}$ czytają w tym samym cyklu zegara kolejne komórki pamięci.

Grupowanie dostępów do pamięci globalnej – zasady



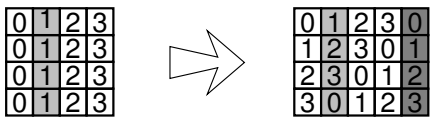
Każdy wątek powinien czytać macierz kolumnami, gdyż wtedy splot wątków czyta fragment jej wiersza spod kolejnych adresów, a taki dostęp zostanie zgrupowany. Jeśli wymagany jest odczyt wierszami, to najpierw trzeba „kafelek” macierzy przepisać do pamięci współdzielonej.



To, w ile dostępów zostanie zgrupowany dany wzór sięgania do pamięci globalnej, zależy od C_c (możliwości obliczeniowych) użytego GPU. Dla $C_c \geqslant 6.0$ dostępy wątków splotu zawsze zostaną zgrupowane w tyle 32 B transakcji ciągłych obszarów, ile potrzeba do przesłania danych splotu.

Konflikty banków pamięci współdzielonej

- Pamięć współdzielona wieloprocesora jest zorganizowana w banki 32-bitowych słów ($32^{2.0}$ lub 16 banków). Słowa o kolejnych adresach znajdują się w kolejnych bankach.
- Splot wątków (lub półsplot) może zrealizować dostęp do tej pamięci równocześnie, o ile każdy wątek sięga do innego banku. W przeciwnym razie występuje konflikt banków i dostępy są serializowane (wielokrotne spowolnienie!).
- Wyjątkiem jest wielokrotny odczyt tego samego słowa (*broadcast*), który nie jest serializowany.
- Jeżeli splot (półsplot) wątków sięga w danej chwili do kolumny tego samego „kafelka”, to ponieważ cała kolumna należy do jednego banku, więc występuje wielokrotny konflikt. Typowym rozwiązaniem jest sztuczne zwiększenie o jeden liczby kolumn w deklaracji macierzy.



Dostępy do pamięci w aplikacji mnożenia macierzy

Dzięki odpowiedniej alokacji pamięci można zagwarantować wyrównanie adresów początkowych wierszy wszystkich macierzy.

1. Kopiowanie
   - Przyrosty indeksów `A` i `B` są równe przyrostom `threadIdx.x`, więc dostępy wątków do pamięci globalnej będą zgrupowane.
   - Przyrosty indeksów `As` i `Bs` są równe przyrostom `threadIdx.x`, więc nie występują konflikty banków pamięci współdzielonej.
2. Obliczenia
   - Indeks `As` nie zależy od `threadIdx.x`, więc wszystkie wątki splotu (*warp*) czytają tę samą komórkę (*broadcast*).
   - Przyrost indeksu `Bs` jest równy przyrostowi `threadIdx.x`, więc nie występują konflikty banków pamięci współdzielonej.

Jądro mnożenia macierzy w CUDA C – matmul4.cu

```
1   for (int i = threadIdx.y + blockIdx.y*blockDim.y; i < m1;
2                              i += gridDim.y*blockDim.y) {
3   for (int j = threadIdx.x + blockIdx.x*blockDim.x; j < n1;
4                              j += gridDim.x*blockDim.x) {
5   float s = 0;
6
7   int idx = threadIdx.y*blockDim.x + threadIdx.x;
8   for (int t = 0; t < pp; t++) // pętla po "kafelkach"
9   {
10    int kx = threadIdx.x + t*blockDim.x;
11    int ky = threadIdx.y + t*blockDim.y;
12    As[idx] = (i < m && kx < p) ? A[i*p + kx] : 0;
13    Bs[idx] = (j < n && ky < p) ? B[ky*n + j] : 0;
14    __syncthreads();
15
16    for (int k = 0; k < blockDim.x; k++)
17      s += As[threadIdx.y*blockDim.y + k] *
18           Bs[k*blockDim.x + threadIdx.x];
19    __syncthreads();
20  }
```

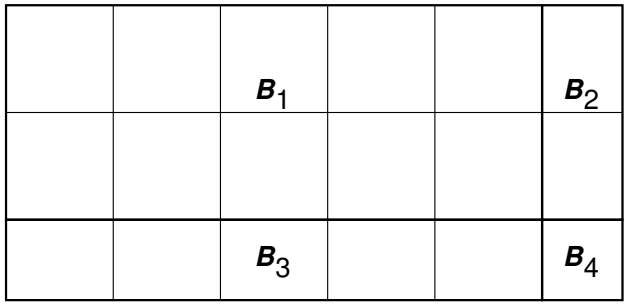
Analiza wydajności implementacji GPU RTX 4070

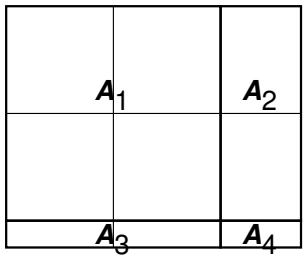
- Do testów przyjmiemy rozmiar „kafelka” $K = M = N = 32$.
- Analizować będziemy tylko czas wykonania samego jądra, bez narzutu na jego uruchomienie i przesyłanie danych.
- GPU3
  `GPU (kernel) time = 1.039 ms (2047.927 GFLOP/s)`
- GPU4
  `GPU (kernel) time = 1.147 ms (1855.841 GFLOP/s)`
- Przyspieszenie wynosi zaledwie 0,91×. Gdzie się podział spodziewany K-krotny zysk?!
  - Kod jądra bardzo się skomplikował (ma 4× więcej linii), więc dłużej się wykonuje i gorzej się optymalizuje.
  - Kod jądra zawiera liczne i zawsze sprawdzane warunki.
  - Skomplikowana arytmetyka adresowa wymaga wielu dodatkowych mnożeń i dodawań, nie uwzględnianych w bilansie FLOP-ów, a obciążających układy SP.
  - Dla $C_c > 7.0$ efektywnie działająca pamięć podręczna, dająca bardzo dobre efekty dla poprzedniej wersji programu.

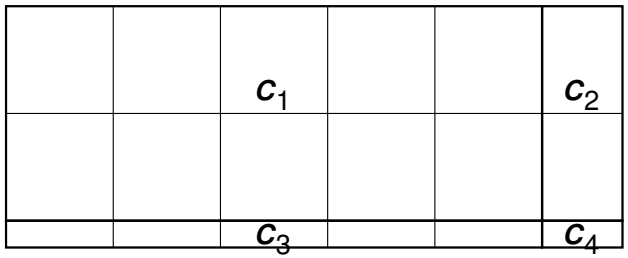
Próba optymalizacji i uproszczenia algorytmu

- Opisane komplikacje kodu są konieczne tylko dla skrajnych „kafelków”, które dla dużych macierzy stanowią znikomy %.
- Ogromną większość pracy może wykonać specjalizowane proste jądro, które zajmuje się „wnętrzem” macierzy.
- Możliwe są dwa podejścia:
  - Macierze blokowe. Wariant stosowany w bibliotekach, np. CUBLAS i MAGMA. Kod jest jeszcze bardziej skomplikowany, ma wiele instrukcji `if` i `switch` uwzględniających „końcówki” o różnych rozmiarach.
  - Macierze rozszerzone. Będziemy mnożyć macierze
    $$\overline{\boldsymbol{C}}_{m_1 \times n_1} = \overline{\boldsymbol{A}}_{m_1 \times p_1} \cdot \overline{\boldsymbol{B}}_{p_1 \times n_1}$$
    o rozmiarach zaokrąglonych w górę do pełnych „kafelków” i uzupełnionych zerami. Kod jądra jest dużo prostszy, a liczba niepotrzebnie wykonywanych mnożeń i dodawań zer – niewielka. Taki wariant przyjmiemy na wykładzie. Dodatkowo wyrównamy każdy wiersz w pamięci:
    $$\overline{\overline{\boldsymbol{C}}}_{m_1 \times n_2}, \overline{\overline{\boldsymbol{A}}}_{m_1 \times p_2}, \overline{\overline{\boldsymbol{B}}}_{p_1 \times n_2}$$

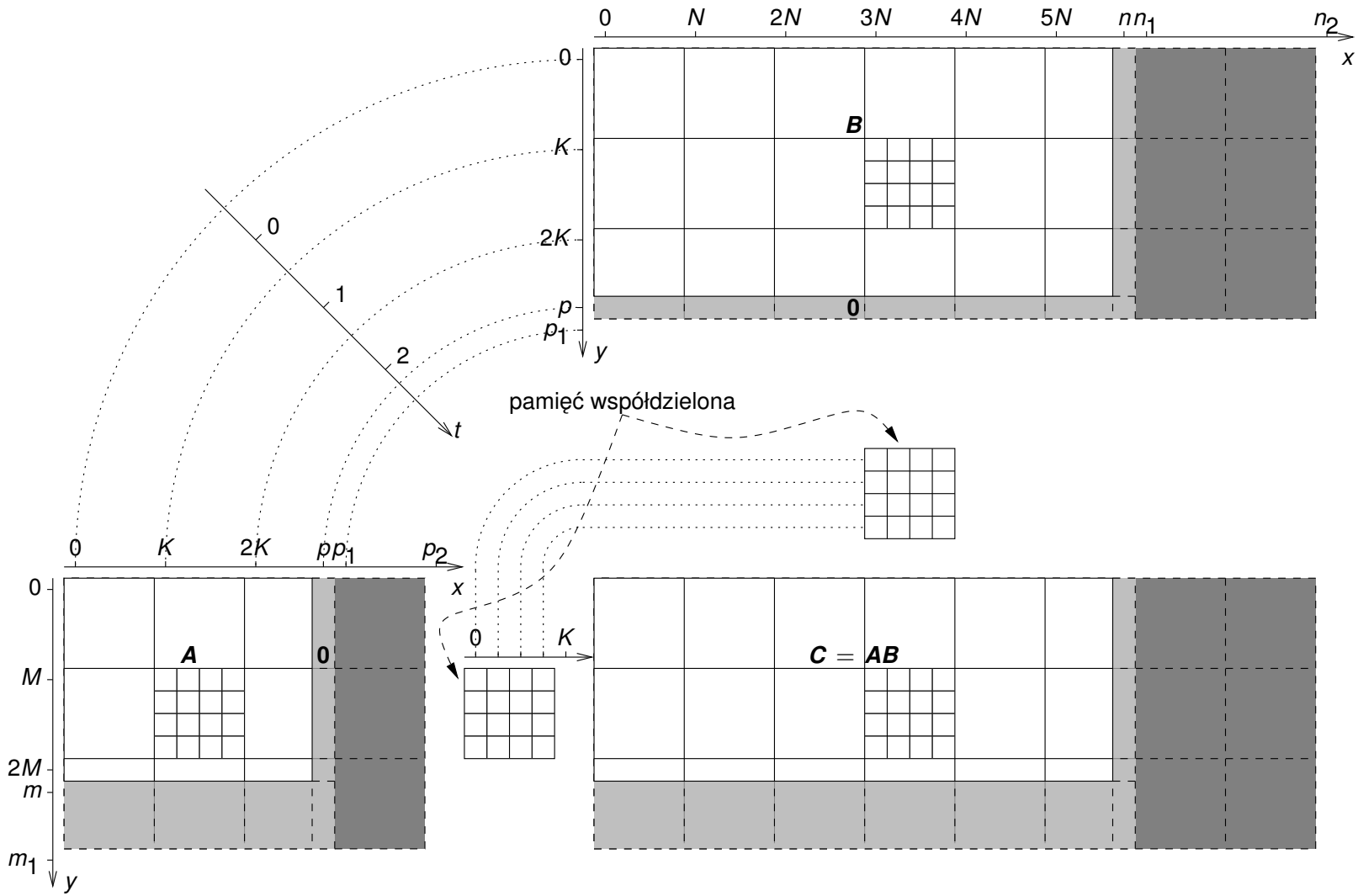
Mnożenie macierzy blokowej techniką „kafelkową”







Mnożenie macierzy rozszerzonej techniką „kafelkową”



Jądro mnożenia macierzy w CUDA C – `matmul5.cu`

```
1  void matmul(...)
2    size_t m1 = ((m + tileSize - 1) / tileSize) * tileSize;
3    ...
4    float *dev_A; size_t A_pitch;
5    checkCudaErrors(cudaMallocPitch(&dev_A, &A_pitch,
6      p1 * sizeof(float), m1));
7    int p2 = A_pitch / sizeof(float);
8    ...
9    checkCudaErrors(cudaMemcpy2D(dev_A,      A_pitch, A,
10     p * sizeof(float), p * sizeof(float), m,
11     cudaMemcpyHostToDevice));
12   checkCudaErrors(cudaMemset2D(dev_A + p, A_pitch, 0,
13     (p1 - p) * sizeof(float), m));
14   ...
15   matmul_kernel<<<dimGrid, dimBlock, shSize>>>
16                 (dev_C, dev_A, dev_B, m1, p1, n1, p2, n2);
17   ...
18   checkCudaErrors(cudaMemcpy2D(C, n * sizeof(float), dev_C,
19     C_pitch, n * sizeof(float), m,
20     cudaMemcpyDeviceToHost));
```

Jądro mnożenia macierzy w CUDA C – matmul5.cu

```
1  __global__
2  static void matmul_kernel(
3    float *C, float *A, float *B, int m1, int p1, int n1,
4                                          int p2, int n2)
5  {
6    extern __shared__ float sh_mem[];
7    float *As = &sh_mem[0];
8    float *Bs = &sh_mem[blockDim.x * blockDim.y];
9
10
11
12
13   for (int i = threadIdx.y + blockIdx.y*blockDim.y; i < m1;
14                              i += gridDim.y*blockDim.y) {
15   for (int j = threadIdx.x + blockIdx.x*blockDim.x; j < n1;
16                              j += gridDim.x*blockDim.x) {
17   float s = 0;
```

Jądro mnożenia macierzy w CUDA C – matmul5.cu

```
1     int idx = threadIdx.y*blockDim.x + threadIdx.x;
2     int kx = threadIdx.x;
3     int ky = threadIdx.y;
4     for (int t = 0; t < p1 / blockDim.x;
5          t++, kx += blockDim.x, ky += blockDim.x)
6     {
7
8
9       As[idx] =                          A[i*p2 + kx];
10      Bs[idx] =                          B[ky*n2 + j];
11      __syncthreads();
12
13      for (int k = 0; k < blockDim.x; k++)
14        s += As[threadIdx.y*blockDim.y + k] *
15             Bs[k*blockDim.x + threadIdx.x];
16      __syncthreads();
17    }
18
19      C[i*n2 + j] = s;
20  }}}
```

Analiza wydajności implementacji GPU RTX 4070

- GPU4
  `GPU (kernel) time = 1.147 ms (1855.841 GFLOP/s)`
- GPU5
  `GPU (kernel) time = 1.122 ms (1898.01 GFLOP/s)`
- I znów przyspieszenie wynosi zaledwie 1,02×. Dlaczego nasze zabiegi tak niewiele pomogły?!
  - Pierwszym problemem jest napisanie zbyt ogólnego programu: dynamiczne zadawanie rozmiaru „kafelka” uniemożliwia kompilatorowi dobrą optymalizację kodu.
  - Ponadto nadal używamy dość skomplikowanych, wymagających mnożeń i dodawań, operacji indeksowych.
- Jak uczynić rozmiar „kafelka” równocześnie statycznym i adaptacyjnym względem możliwości konkretnego procesora GPU? Użycie preprocesora języka C (dyrektywa **`#define`**) nie wchodzi w grę. Wykorzystamy mechanizm szablonów (*template*) funkcji języka C++.

Jądro mnożenia macierzy w CUDA C – matmul6.cu

```
 1  void matmul(float *C, float *A, float *B,
 2              int m, int p, int n, ...)
 3  {
 4    ...
 5    int devID;
 6    checkCudaErrors(cudaGetDevice(&devID));
 7    cudaDeviceProp props;
 8    checkCudaErrors(cudaGetDeviceProperties(&props, devID));
 9    int tileSize = (props.major < 2) ? 16 : 32;
10    ...
11
12    if (tileSize == 16)
13      matmul_kernel<16><<<dimGrid, dimBlock>>>
14                  (dev_C, dev_A, dev_B, m1, p1, n1, p2, n2);
15    else
16      matmul_kernel<32><<<dimGrid, dimBlock>>>
17                  (dev_C, dev_A, dev_B, m1, p1, n1, p2, n2);
18    ...
19  }
```

Jądro mnożenia macierzy w CUDA C – matmul6.cu

```
1  template <int K>
2  __global__
3  static void matmul_kernel(
4    float *C, float *A, float *B, int m1, int p1, int n1,
5                                          int p2, int n2)
6  {
7
8
9
10   ...
11   __shared__ float As[K * K], Bs[K * K];
```

Analiza wydajności implementacji GPU RTX 4070

- GPU5
  `GPU (kernel) time = 1.122 ms (1898.01 GFLOP/s)`
- GPU6
  `GPU (kernel) time = 0.734 ms (2902.049 GFLOP/s)`
- Kolejne przyspieszenie wynosi 1,53×. Już zbliżamy się do 1/10 dysponowanej mocy obliczeniowej GPU...
- Teraz, gdy liczba powtórzeń pętli obliczania cząstkowego iloczynu skalarnego jest statyczna, mogliśmy agresywnie zastosować technikę rozwijania pętli:

```
1    #pragma unroll
2    for (int k = 0; k < K; k++) { ... }
```

Dla starych kart o $C_c < 2.0$, gdzie „kafelek” miał rozmiar $K = 16$, kompilator sam rozwijał tę pętlę.

Technika rozwijania pętli

Porównanie pętli obliczania iloczynu skalarnego w dwóch ostatnich implementacjach algorytmu:

`matmul5.sass`

```
{ ISCADD R28, R3.reuse, R25, 0x2;
  LDS.U.32 R4, [R25]                }
{ IADD32I R5, R5, 0x4;
  LDS.U.32 R26, [R2]                }
{ ISCADD R30, R3, R28, 0x2;
  LDS.U.32 R27, [R2+0x4]            }
{ ISETP.GE.U32.AND P0, PT, R5, ...
  LDS.U.32 R24, [R28]               }
  LDS.U.32 R29, [R2+0xc];
{ FFMA R26, R4, R26, R23;
```

Instrukcja obliczeniowa (`FFMA`) zajmuje 10% kodu. Reszta to narzut na obliczanie adresów i zapętlenie kodu.

`matmul6.sass`

```
{ FFMA R5, R24, R5, R27;
  LDS.U.32 R24, [R12+0x240] }
{ FFMA R6, R28, R6, R5;
  LDS.U.32 R27, [R12+0x280] }
{ FFMA R7, R25, R7, R6;
  LDS.U.32 R28, [R12+0x2c0] }
{ FFMA R26, R26, R8, R7;
  LDS.U.32 R25, [R12+0x300] }
```

Instrukcja obliczeniowa (`FFMA`) zajmuje 50% kodu. Adresy (offsety) są ustalone i nie muszą być wyliczane. Pętli nie ma, gdyż została rozwinięta.

Kod ostatniej wersji algorytmu wygląda już całkiem dobrze.

Czy nasz program da się jeszcze zoptymalizować?

- Pierwotny kod CPU (z rozwiniętym w miejscu jądrem):

```
1  int i, j, k;
2  for (i = 0; i < m; i++)
3    for (j = 0; j < n; j++) {
4      float s = 0;
5      for (k = 0; k < p; k++)
6        s += A[i*p + k] * B[k*n + j];
7      C[i*n + j] = s;
8    }
```

- Równoważny kod CPU pokazujący niezależność pętli:

```
1  int i, j, k;
2  memset(C, 0, m * n * sizeof(float));
3  for (i = 0; i < m; i++)
4    for (j = 0; j < n; j++)
5      for (k = 0; k < p; k++)
6        C[i*n + j] += A[i*p + k] * B[k*n + j];
```

- Skoro trzy pętle są niezależne, to można je permutować.

Wpływ zmiany kolejności pętli mnożenia macierzy

kolejność	czas [ms]
`ijk`	1746
`ikj`	**342**
`jik`	1496
`jki`	10272
`kij`	376
`kji`	11445

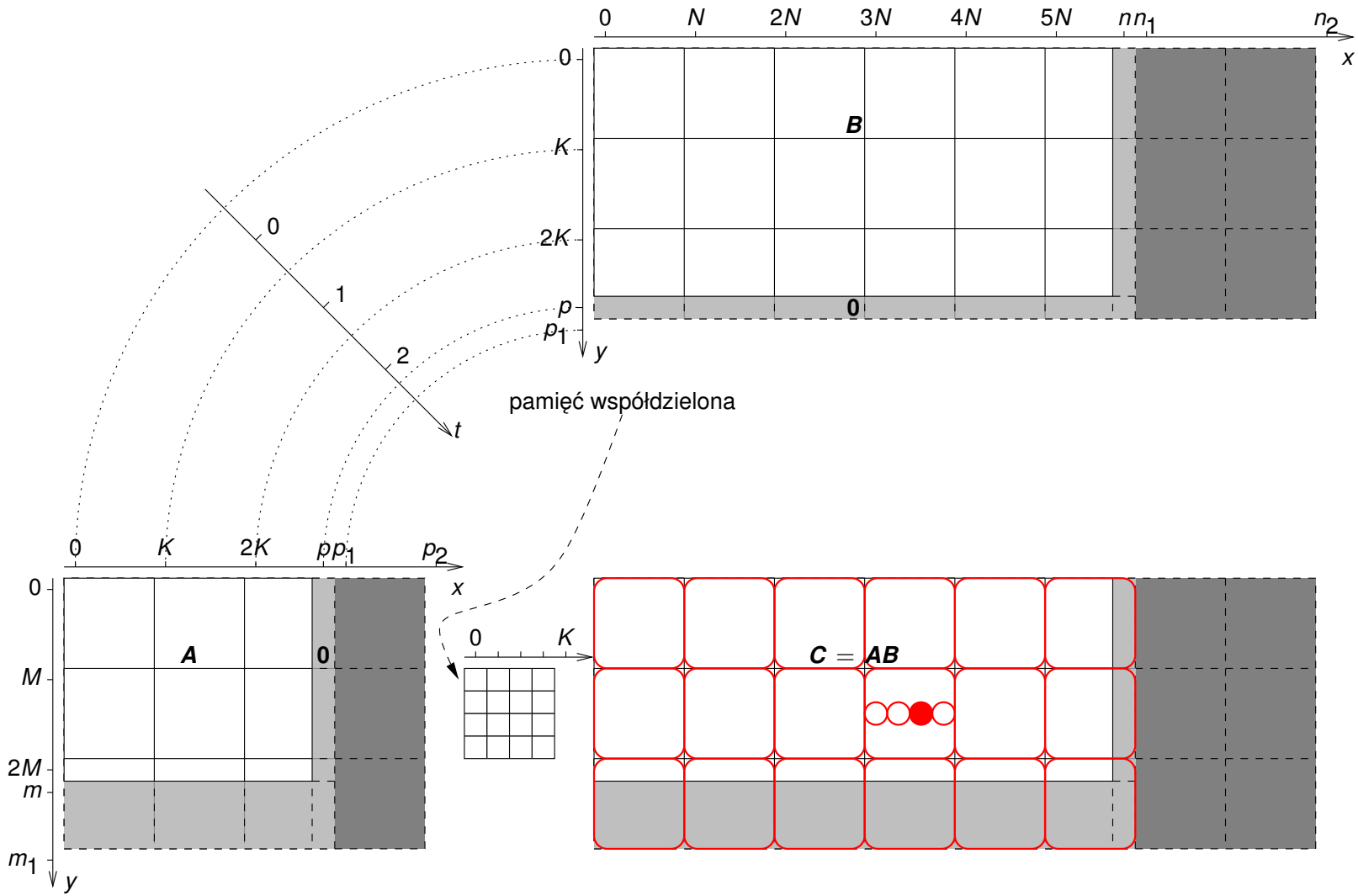
- Różnica między skrajnymi przypadkami wynosi aż 33$\times$!
- Najlepsze wyniki daje kolejność `ikj` lub `kij`, przy której w najbardziej wewnętrznej pętli sięgamy wciąż do tego samego elementu macierzy $\boldsymbol{A}$ oraz do kolejnych elementów wierszy macierzy $\boldsymbol{B}$ i $\boldsymbol{C}$, umieszczonych w kolejnych komórkach pamięci. Utrzymujemy wtedy lokalność dostępów do pamięci (por. `Cachegrind`).
- Użyty przez nas „szkolny” algorytm zachowuje się źle.

Algorytm Volkova

V. Volkov, J. W. Demmel, *Benchmarking GPUs to Tune Dense Linear Algebra*
Proc. 2008 ACM/IEEE Conf. on Supercomputing, Piscataway, NJ, USA

- Dotychczasowe programy obliczały $m \times n$ iloczynów skalarnych (*inner product*) p-elementowych wektorów: i-tego wiersza $\boldsymbol{A}$ przez j-tą kolumnę $\boldsymbol{B}$. Każdy iloczyn był jednym elementem $\boldsymbol{C}$ (skalarem) i obliczał go jeden wątek.
- Volkov zaproponował sumowanie p macierzy $\boldsymbol{C}_k$, będących iloczynem macierzowym pierwszego rzędu (*outer product*) k-tej kolumny $\boldsymbol{A}$ przez k-ty wiersz $\boldsymbol{B}$. Jeden wątek oblicza całą kolumnę „kafelka” macierzy $\boldsymbol{C}$.
- Macierz $\boldsymbol{A}$ pobieramy do pamięci współdzielonej „kafelkami” $M \times K$. Ponieważ cała kolumna „kafelka” $\boldsymbol{A}$ będzie mnożona przez ten sam element $\boldsymbol{B}$, więc macierzy $\boldsymbol{B}$ nie trzeba „kafelkować” i przepisywać do pamięci współdzielonej – jedyny potrzebny wątkowi element $\boldsymbol{B}$ można przechować w rejestrze (szybkim!).
- W rejestrach można też przechowywać sumy częściowe kolumny „kafelka” $\boldsymbol{C}$ – szybki dostęp, oszczędność *shmem*!

Mnożenie macierzy algorytmem Volkova



Jądro mnożenia macierzy w CUDA C – matmul7.cu

```
1  #define M 16 // potęga 2
2  #define K 32 // potęga 2
3  #define N 64 // potęga 2, blockDim.x, >= 32
4  __global__
5  static void matmul_kernel(
6    float *C, float *A, float *B, int m1, int p1, int n1,
7                                          int p2, int n2)
8  {
9    for (int i =                  blockIdx.y*M; i < m1;
10                             i += gridDim.y*M) {
11   for (int j = threadIdx.x + blockIdx.x*N; j < n1;
12                             j += gridDim.x*N) {
13   float s[M]; for (int r = 0; r < M; r++) s[r] = 0;
14
15
16
17
18
19
20     ...
```

Jądro mnożenia macierzy w CUDA C – matmul7.cu

```
1     for (int t = 0; t < p1 / K;
2          t++)
3     {
4       __shared__ float As[M * K];
5       for (int r = 0; r < M; r += N / K)      // <- nowa pętla!
6         As[  r * K +  threadIdx.x] =
7           A[(i + r + (threadIdx.x / K))*p2 +
8             (t * K + (threadIdx.x % K))];
9       __syncthreads();
10
11      float *Bptr = B + t*K*n2 + j;
12  #pragma unroll
13      for (int k = 0; k < K; k++, Bptr += n2)
14        for (int r = 0; r < M; r++)           // <- nowa pętla!
15          s[r] += As[r*K + k] * *Bptr;
16      __syncthreads();
17    }
18    float *Cptr = C + i*n2 + j;
19    for (int r = 0; r < M; r++, Cptr += n2)   // <- nowa pętla!
20      *Cptr = s[r];  }}}
```

Jądro mnożenia macierzy w CUDA C – matmul7.cu

```
1  void matmul(float *C, float *A, float *B,
2              int m, int p, int n, ...)
3  { ...
4
5    size_t m1 =  (m + M       - 1) & ~(M - 1); // wyrównanie
6    size_t p1 =  (p + K       - 1) & ~(K - 1); // w górę (M, K,
7    size_t n1 =  (n + N       - 1) & ~(N - 1); // N - potęgi 2)
8    ...
9    dim3 dimGrid(BGx, BGy), dimBlock(N); // <- mniej wątków/blok
10   ...
11   matmul_kernel<<<dimGrid, dimBlock>>>
12                (dev_C, dev_A, dev_B, m1, p1, n1, p2, n2);
13   ...
14 }
```

Analiza wydajności algorytmu Volkova RTX 4070

- GPU6
  `GPU (kernel) time = 0.734 ms (2902.049 GFLOP/s)`
- GPU7
  `GPU (kernel) time = 0.309 ms (6883.357 GFLOP/s)`
- Kolejne przyspieszenie: 2,37×. Osiągnęliśmy już 1/5 dysponowanej mocy obliczeniowej GPU.
- Główne cechy charakterystyczne algorytmu Volkova:
  - Zwiększona „ziarnistość” równoległości: jeden wątek liczy nie jeden element macierzy, tylko całą kolumnę jej „kafelka”.
  - Blok zawiera mniej wątków, wykonujących więcej pracy.
  - Opóźnienie dostępów do pamięci ukrywamy nie wieloma wątkami, tylko buforując dane w pamięci współdzielonej.
  - Rejestry zostały przeznaczone nie na dublowanie (tych samych we wszystkich wątkach) informacji pomocniczych (np. adresów), tylko na istotne dane obliczeniowe (`s[M]`).
  - Mniejsze wykorzystanie zasobów, ale większa wydajność.
- Wątek sięga do nie kolejnych komórek *shmem*.

Jądro mnożenia macierzy w CUDA C – matmul8.cu

```
1  #define M1 (M + 1) // trik jak przy konfliktach banków shmem
2  __global__ static void matmul_kernel(float *C, float *A,
3    float *B, int m1, int p1, int n1, int p2, int n2)
4  { ...
5    for (int t = 0; t < p1 / K; t++)
6    {
7      __shared__ float AsT[K * M1];
8      for (int r = 0; r < M; r += N / K)
9        AsT[      r + (threadIdx.x / K) +
10                M1 * (threadIdx.x % K)]  =
11         A[(i + r + (threadIdx.x / K))*p2 +
12           (t * K + (threadIdx.x % K))];
13     __syncthreads();
14
15     float *Bptr = B + t*K*n2 + j;
16 #pragma unroll
17     for (int k = 0; k < K; k++, Bptr += n2)
18       for (int r = 0; r < M; r++)
19         s[r] += AsT[k*M1 + r] * *Bptr;
20     __syncthreads(); ...
```

Analiza wydajności kodu GPU RTX 4070

- GPU7
  `GPU (kernel) time = 0.309 ms (6883.357 GFLOP/s)`
- GPU8
  `GPU (kernel) time = 0.287 ms (7424.192 GFLOP/s)`
- I znów uzyskaliśmy pewne przyspieszenie: 1,08×.
- Porównajmy „kamienie milowe” rozwoju programu:

wersja	czas [ms]
CPU	48,761
„naiwna” GPU	1,039
optymalizowana GPU	0,287

- „Pętla” obliczania iloczynu skalarnego `matmul8.sass`:

```
FFMA R12, R55.reuse, R12, R57;
```

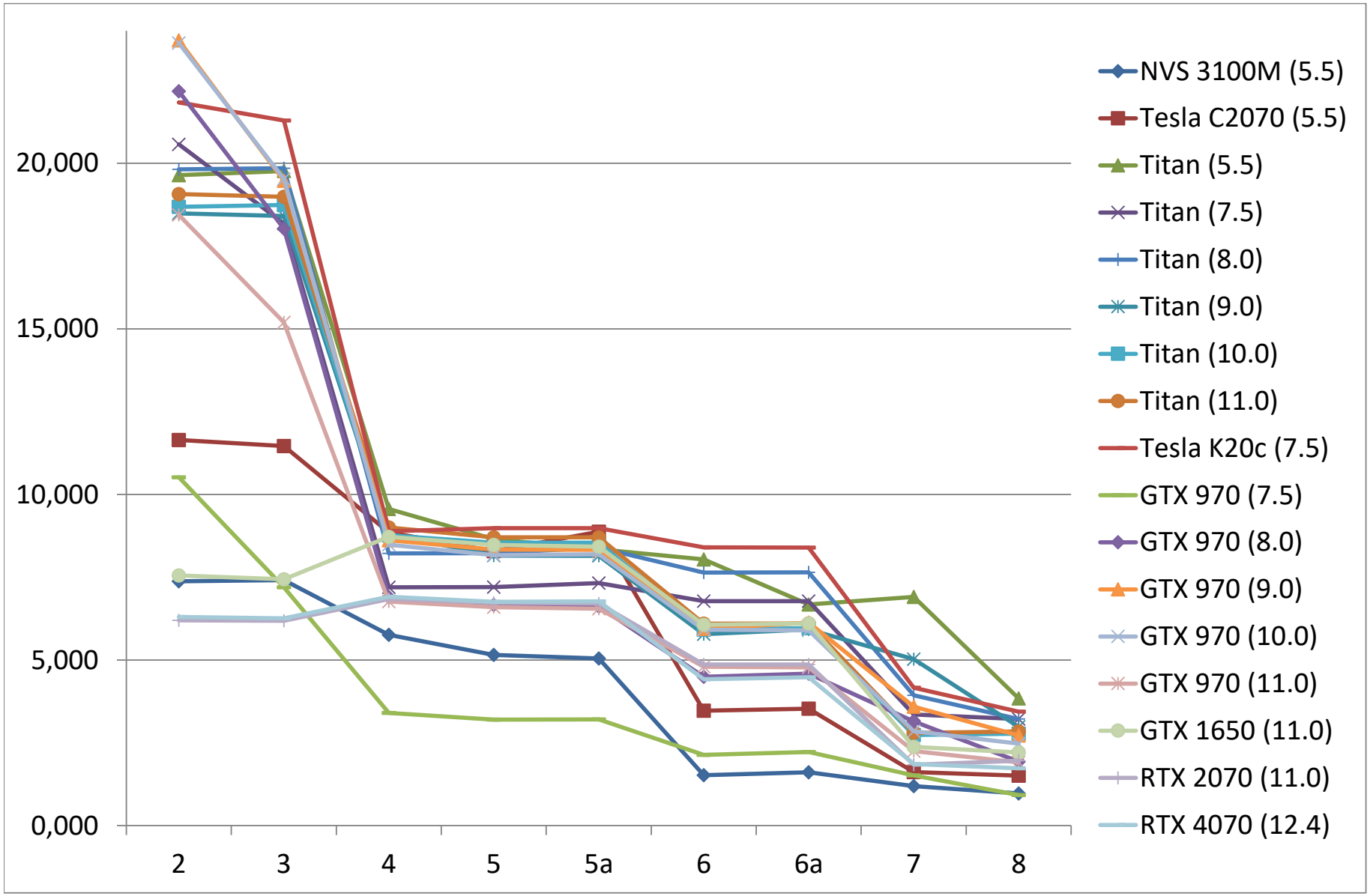
  Instrukcja obliczeniowa (`FMAD`) zajmuje ≈100% kodu.
  Bardzo mały narzut na obliczanie adresów i zapętlenie.
- Pozostaje sprawdzić, jakie wyniki uzyskują „najlepsi”...

Jądro mnożenia macierzy w CUDA C – matmulC.cu

```
1  void matmul(float *C, float *A, float *B,
2              int m, int p, int n, ...)
3  { ...
4    cublasHandle_t h;
5    CUBLAS_SAFE_CALL(cublasCreate(&h));
6    const float alpha = 1.0f;
7    const float beta  = 0.0f;
8    ...
9    CUBLAS_SAFE_CALL(cublasSgemm(h, CUBLAS_OP_N, CUBLAS_OP_N,
10     n, m, p, &alpha, dev_B, n, dev_A, p, &beta, dev_C, n));
11   ...
12   CUBLAS_SAFE_CALL(cublasDestroy(h));
13   ...
14 }
```

- GPU8 – % dysponowanej mocy obliczeniowej RTX 4070
  `GPU (kernel) time = 0.287 ms (7424.192 GFLOP/s)`
- GPU-CUBLAS
  `GPU (kernel) time = 0.166 ms (12905.551 GFLOP/s)`
- Jesteśmy aktualnie 1,74 $\times$ gorsi od „najlepszych”, ale...

Wydajność (względem CUBLAS) na różnych kartach



Mnożenia macierzy w CUDA C RTX 4070

Choć biblioteka CUBLAS wydaje się bezkonkurencyjna...

- GPU8

```
GPU (kernel) time = 0.287 ms (7424.192 GFLOP/s)
```

- GPU-CUBLAS

```
GPU (kernel) time = 0.166 ms (12905.551 GFLOP/s)
```

...to jednak jej wykorzystanie ma swój koszt:

- GPU8

```
CPU (total!) time = 48.688 ms (43.721 GFLOP/s)
```

- GPU-CUBLAS

```
CPU (total!) time = 57.939 ms (36.740 GFLOP/s)
```