Mnożenie macierzy rzeczywistych

- Problem: $\boldsymbol{C}_{m\times n} = \boldsymbol{A}_{m\times p} \cdot \boldsymbol{B}_{p\times n}$, gdzie $a_{ik}, b_{kj}, c_{ij} \in \mathbb{R}$.
- Każdy element macierzy wynikowej $\boldsymbol{C}$ jest iloczynem skalarnym wiersza $\boldsymbol{A}$ przez kolumnę $\boldsymbol{B}$ (indeksy od 0):

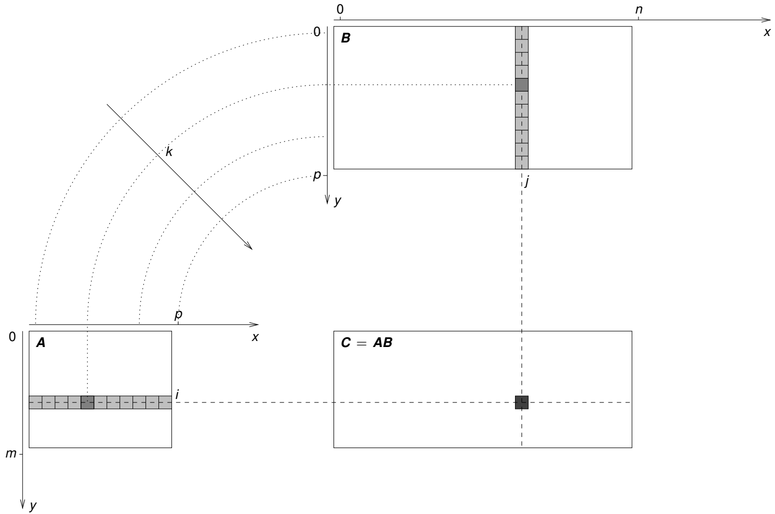
$$c_{ij} = \sum_{k=0}^{p-1} a_{ik} b_{kj}, \qquad i = 0, \ldots, m-1, \quad j = 0, \ldots, n-1$$

- Czy warto się tym problemem zajmować? Czy potrafimy dostarczyć procesorowi GPU dostatecznie dużo pracy?

Złożoność obliczeniowa ($\cdot$ i $+$)	$mnp + mn(p-1)$	[FLOP]
Liczba dostępów do pamięci	$mn + mp + pn$	[T]

- Współczynnik pamięciowy złożoności obliczeniowej CGMA (*Compute to Global Memory Access ratio*) dla $m = p = n$ wynosi $\frac{2}{3}n - \frac{1}{3}$ i dla $n \to \infty$ może być dostatecznie duży.
- Uwaga! Dla algorytmu iloczynu skalarnego współczynnik CGMA wynosi już tylko $\frac{p+(p-1)}{1+p+p} = \frac{2p-1}{2p+1} \approx 1$.

Podstawowy schemat mnożenia macierzy



Generacja macierzy w MATLAB-ie – `matgen.m`

```
function matgen(m, p, n)

%% utworzenie losowych macierzy czynników i iloczynu
A = randn(m, p);
B = randn(p, n);
C = A * B;

%% zapamiętanie rozmiarów i macierzy w pliku binarnym
f = fopen('matmul.dat', 'wb');
fwrite(f, [m, p, n], 'int');
% zapamiętujemy transponowane macierze, bo język C
% przechowuje macierze wierszami, a nie kolumnami
fwrite(f, A.', 'single');
fwrite(f, B.', 'single');
fwrite(f, C.', 'single');
fclose(f);
```

Szkielet mnożenia macierzy w języku C – matmul.c

```c
#include <stdlib.h>
#include <stdio.h>

#include "matmul.h"

int main(int argc, char *argv[])
{
  app_timer_t start, stop;
  int         m,  p,  n;
  float               *A, *B, *C, *D;
  read_data(&m, &p, &n, &A, &B, &C, &D);
  timer(&start);
  matmul(C, A, B, m, p, n, argc, argv);
  timer(&stop);
  elapsed_time(start, stop, 2 * m * p * n);
  matcmp(C, D, m, n);
  free_mem(A, B, C, D);
  if (IsDebuggerPresent()) getchar();
  return 0;
}
```

Szkielet mnożenia macierzy w języku C – `matmul.c`

```
void read_data(int *m_ptr, int *p_ptr, int *n_ptr,
               float **A_ptr, float **B_ptr,
               float **C_ptr, float **D_ptr)
{
  FILE *f = fopen("matmul.dat", "rb");

  fread(m_ptr, sizeof(int), 1, f);
  fread(p_ptr, sizeof(int), 1, f);
  fread(n_ptr, sizeof(int), 1, f);

  alloc_mem(*m_ptr, *p_ptr, *n_ptr,
            A_ptr, B_ptr, C_ptr, D_ptr);

  fread(*A_ptr, sizeof(float), *m_ptr * *p_ptr, f);
  fread(*B_ptr, sizeof(float), *p_ptr * *n_ptr, f);
  fread(*D_ptr, sizeof(float), *m_ptr * *n_ptr, f);

  fclose(f);
}
```

Szkielet mnożenia macierzy w języku C – `matmul.c`

```c
void alloc_mem(int m, int p, int n,
               float **A_ptr, float **B_ptr,
               float **C_ptr, float **D_ptr)
{
  *A_ptr = (float *) malloc(m * p * sizeof(float));
  *B_ptr = (float *) malloc(p * n * sizeof(float));
  *C_ptr = (float *) malloc(m * n * sizeof(float));
  *D_ptr = (float *) malloc(m * n * sizeof(float));
}

void free_mem(float *A, float *B, float *C, float *D)
{
  free(A);
  free(B);
  free(C);
  free(D);
}
```

Szkielet mnożenia macierzy w języku C – `matmul.c`

```c
#define WINDOWS_LEAN_AND_MEAN
#include <windows.h>

typedef LARGE_INTEGER app_timer_t;

#define timer(t_ptr) QueryPerformanceCounter(t_ptr)

void elapsed_time(app_timer_t start, app_timer_t stop,
                  unsigned long flop)
{
  double etime;
  LARGE_INTEGER clk_freq;
  QueryPerformanceFrequency(&clk_freq);
  etime = (stop.QuadPart - start.QuadPart) /
          (double) clk_freq.QuadPart;
  printf("CPU␣(total!)␣time␣=␣%.3f␣ms␣(%6.3f␣GFLOP/s)\n",
         etime * 1e3, 1e-9 * flop / etime);
}
```

Szkielet mnożenia macierzy w języku C – `matmul.c`

```c
#include <math.h>

void matcmp(float *C, float *D, int m, int n)
{
  int k;
  float d, e = -1.0f;
  for (k = 0; k < m * n; k++)
    if ((d = fabsf(C[k] - D[k])) > e)
      e = d;
  printf("max. abs. err. = %.1e\n", e);
}
```

Sprawdzanie poprawności

Wydajność programu nie ma żadnego znaczenia, jeżeli daje on niepoprawne wyniki!

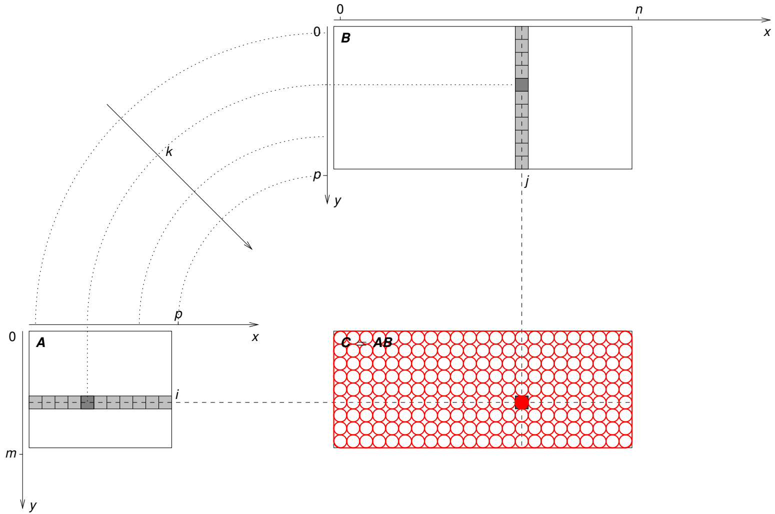
Jądro mnożenia macierzy w języku C – matmul0.c

```
#include "matmul.h"

void matmul(float *C, float *A, float *B,
            int m, int p, int n, ...)
{
  int i, j;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      matmul_kernel(i, j, C, A, B, m, p, n);
}

static void matmul_kernel(int i, int j,
  float *C, float *A, float *B, int m, int p, int n)
{
  int k;
  float s = 0;
  for (k = 0; k < p; k++)
    s += A[i*p + k] * B[k*n + j];
  C[i*n + j] = s;
}
```

Podstawowy schemat mnożenia macierzy – „ziarna”



Mnożenie macierzy w języku CUDA C – `matmul1.cu`

```
#include "helper_cuda.h" // -I$(NVCUDASAMPLES_ROOT)/common/inc
#include "matmul.h"

void matmul(float *C, float *A, float *B,
            int m, int p, int n, ...)
{
  // 1. Wybór urządzenia CUDA dla kolejnych funkcji API
  checkCudaErrors(cudaSetDevice(0));

  // 2. Alokacja pamięci urządzenia
  float *dev_A, *dev_B, *dev_C;
  checkCudaErrors(cudaMalloc(&dev_A, m*p*sizeof(float)));
  checkCudaErrors(cudaMalloc(&dev_B, p*n*sizeof(float)));
  checkCudaErrors(cudaMalloc(&dev_C, m*n*sizeof(float)));

  // 3. Kopiowanie danych gospodarza do urządzenia
  checkCudaErrors(cudaMemcpy(dev_A, A, m*p*sizeof(float),
                  cudaMemcpyHostToDevice));
  checkCudaErrors(cudaMemcpy(dev_B, B, p*n*sizeof(float),
                  cudaMemcpyHostToDevice));
```

Mnożenie macierzy w języku CUDA C – `matmul1.cu`

```
// 4. Asynchroniczne uruchomienie jądra
dim3 dimBlock(n, m);

cudaEvent_t start, stop; // pomiar czasu wykonania jądra
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));
checkCudaErrors(cudaEventRecord(start, 0));

    matmul_kernel<<<1, dimBlock>>>
                (dev_C, dev_A, dev_B, m, p, n);

checkCudaErrors(cudaGetLastError());

checkCudaErrors(cudaEventRecord(stop, 0));
checkCudaErrors(cudaEventSynchronize(stop));
float elapsedTime;
checkCudaErrors(cudaEventElapsedTime(&elapsedTime,
                                     start, stop));
checkCudaErrors(cudaEventDestroy(start));
checkCudaErrors(cudaEventDestroy(stop));
```

Mnożenie macierzy w języku CUDA C – `matmul1.cu`

```
    // 5. Synchronizacja z jądrem (bariera)
    checkCudaErrors(cudaDeviceSynchronize());

    // 6. Kopiowanie wyników pracy z urządzenia do gospodarza
    checkCudaErrors(cudaMemcpy(C, dev_C, m*n*sizeof(float),
                    cudaMemcpyDeviceToHost));

    // 7. Zwolnienie pamięci urządzenia
    checkCudaErrors(cudaFree(dev_C));
    checkCudaErrors(cudaFree(dev_B));
    checkCudaErrors(cudaFree(dev_A));

    checkCudaErrors(cudaDeviceReset()); // dla debuggera

    printf("GPU (kernel) time = %.3f ms (%6.3f GFLOP/s)\n",
           elapsedTime, 2e-6 * m * n * p / elapsedTime);
}
```

Jądro mnożenia macierzy w CUDA C – matmul1.cu

```
__global__
static void matmul_kernel(
  float *C, float *A, float *B, int m, int p, int n)
{
  int i = threadIdx.y;
  int j = threadIdx.x;
  int k;
  float s = 0;
  for (k = 0; k < p; k++)
    s += A[i*p + k] * B[k*n + j];
  C[i*n + j] = s;
}
```

Położenie „ziarna”, nad którym pracuje dany wątek. . .

. . . ustalamy na podstawie współrzędnych wątku w bloku i współrzędnych bloku w sieci.

Porównanie wydajności implementacji CPU i GPU

- Na początek prosty test dla $m = 13$, $p = 24$, $n = 35$.
- CPU (AMD Ryzen 9 7900)

```
CPU (total!) time = 0.005 ms ( 4.325 GFLOP/s)
max. abs. err. = 2.9e-06
```

- GPU (GeForce RTX 4070: 56 SM, 7168 rdzeni, $C_c = 8.9$)

```
GPU (kernel) time = 0.024 ms (0.927 GFLOP/s)
CPU (total!) time = 52.35 ms (0.000 GFLOP/s)
```

- Obie implementacje dały prawidłowe wyniki.
- Nawet czas wykonania samego jądra GPU (nie mówiąc o narzutach na transfer danych i uruchomienie jądra) jest znacznie dłuższy niż całkowity czas dla CPU.
- Wiadomo, że implementacja GPU nie jest efektywna – wykorzystuje tylko jeden blok, więc tylko jeden SM.
- Czasy są bardzo krótkie – zwiększymy rozmiar problemu, aby zmniejszyć narzuty i zwiększyć dokładność pomiaru.

Porównanie wydajności implementacji CPU i GPU

- Do dalszych testów przyjmiemy $m = p = n = 1021$.
- CPU

  ```
  CPU (total!) time = 572.073 ms (3.72 GFLOP/s)
  max. abs. err. = 2.1e-04
  ```

- GPU (GeForce RTX 4070)

  ```
  CUDA error at ../matmul1.cu:52
    code=9(cudaErrorInvalidConfiguration)
    "cudaGetLastError()"
  ```

  Błąd oznacza nieprawidłową konfigurację wykonania (*execution configuration*), opisywaną składnią `<<<...>>>`.
- Przyczyna niepowodzenia:

$$T_B = mn = 1021^2 > T_{B\,\mathrm{max}} = 1024$$

- Podzielimy wątki pomiędzy wiele bloków, odpowiadających „kafelkom” (*tile*) o rozmiarach $M \times N$ macierzy $\boldsymbol{C}$.

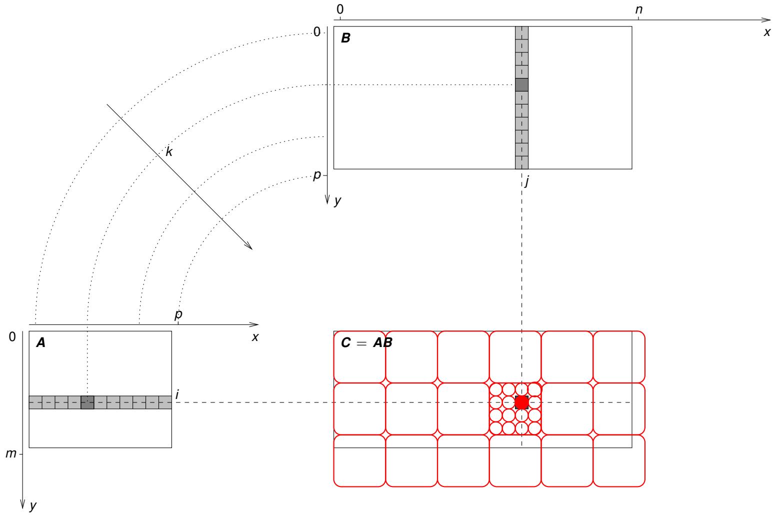
Parametry „geometryczne” CUDA dla różnych C_c

Parametr	`cudaDeviceGetAttribute` →`cudaDevAttrMax...`	*1.x*	*2.x*	3.x	5.x 6.x	7.x 8.x
$B_{G,x\ \mathtt{max}}$	`...GridDimX`	$2^{16}-1$		$2^{31}-1$		
$B_{G,y\ \mathtt{max}}$	`...GridDimY`	$2^{16}-1$				
$B_{G,z\ \mathtt{max}}$	`...GridDimZ`	1	$2^{16}-1$			
$T_{B,x\ \mathtt{max}}$	`...BlockDimX`	512	1024			
$T_{B,y\ \mathtt{max}}$	`...BlockDimY`	512	1024			
$T_{B,z\ \mathtt{max}}$	`...BlockDimZ`	64				
$T_{B\ \mathtt{max}}$	`...ThreadsPerBlock`	512	1024			

Legenda (J_K oznacza liczbę J-ów na jeden K):

- G sieć bloków (*grid*)
- B blok wątków (*block*)
- T wątek (*thread*)
- x,y,z współrzędne „kierunkowe”

Schemat mnożenia macierzy z podziałem *C* na bloki



Jądro mnożenia macierzy w CUDA C – `matmul2.cu`

```
void matmul(float *C, float *A, float *B,
            int m, int p, int n, ...)
{ ...
  dim3 dimBlock(N, M), dimGrid((n+N-1)/N, (m+M-1)/M);
  matmul_kernel<<<dimGrid, dimBlock>>>
                  (dev_C, dev_A, dev_B, m, p, n);
  ...
}
__global__
static void matmul_kernel(
  float *C, float *A, float *B, int m, int p, int n)
{
  int i = threadIdx.y + blockIdx.y*blockDim.y; if (i < m) {
  int j = threadIdx.x + blockIdx.x*blockDim.x; if (j < n) {
  float s = 0;
  for (int k = 0; k < p; k++)
    s += A[i*p + k] * B[k*n + j];
  C[i*n + j] = s;
}}}
```

Porównanie wydajności implementacji CPU i GPU

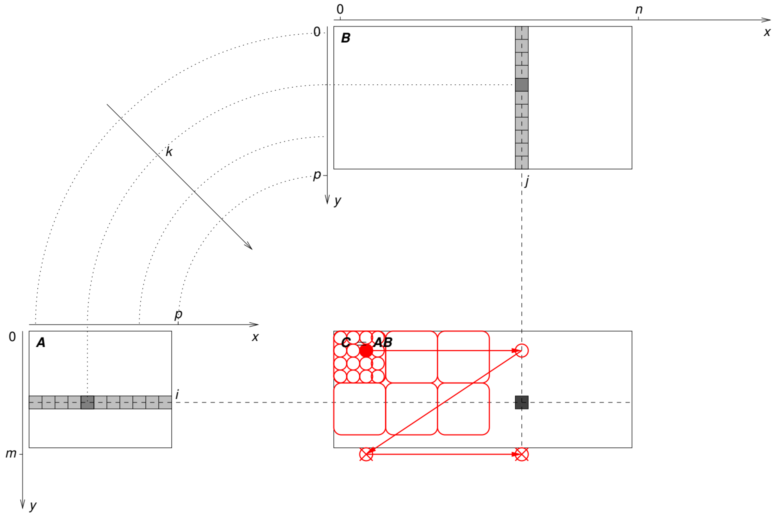
- Do dalszych testów przyjmiemy $M = N = 32$. Tym razem spełnione jest już ograniczenie $T_B = MN \leqslant T_{B\ \max}$.
- GPU (GeForce RTX 4070)

```
GPU (kernel) time = 1.039 ms (2047.927 GFLOP/s)
CPU (total!) time = 48.761 ms (43.655 GFLOP/s)
```

- Jeśli m (albo n) nie jest wielokrotnością M (odpowiednio N), to musimy pokryć „kafelkami” macierz z nadmiarem...
- ...i dlatego wyniki są prawidłowe dopiero po ograniczeniu dziedziny obliczeń – inaczej dodajemy przypadkowe wartości i sięgamy poza przydzieloną pamięć.
- Bez trudu uzyskaliśmy przyspieszenie $550\times$ lub $12\times$.
- Kod nadal nie jest w pełni uniwersalny ze względu na ograniczenia $B_{G,x} \leqslant B_{G,x\ \max}$ i $B_{G,y} \leqslant B_{G,y\ \max}$, czyli:

$$mn \leqslant B_{G,x\ \max} B_{G,y\ \max} T_{B\ \max} \stackrel{3.0}{=} 1{,}4 \cdot 10^{17}.$$

Mnożenie macierzy o dużych (dowolnie!) rozmiarach



Jądro mnożenia macierzy w CUDA C – `matmul3.cu`

```
void matmul(float *C, float *A, float *B,
            int m, int p, int n, ...)
{ ...
  dim3 dimBlock(N, M), dimGrid(gridSizeX, gridSizeY);
  ...
}
__global__
static void matmul_kernel(
  float *C, float *A, float *B, int m, int p, int n)
{
  for (int i = threadIdx.y + blockIdx.y*blockDim.y;     i<m;
                        i += gridDim.y*blockDim.y) {
  for (int j = threadIdx.x + blockIdx.x*blockDim.x;     j<n;
                        j += gridDim.x*blockDim.x) {
  float s = 0;
  for (int k = 0; k < p; k++)
    s += A[i*p + k] * B[k*n + j];
  C[i*n + j] = s;
}}}
```

Analiza wydajności implementacji GPU RTX 4070

- Program działa poprawnie dla dowolnych rozmiarów $\boldsymbol{C}$.
- Powróciliśmy do pierwotnego kodu zawierającego trzy zagnieżdżone pętle. Indeksy pętli zewnętrznych zwiększamy tym razem nie o 1, tylko o liczbę wątków wzdłuż danego wymiaru, równą $T_{B,x}B_{G,x}$ ($T_{B,y}B_{G,y}$).
- Każdy wątek oblicza wiele elementów $\boldsymbol{C}$ – przyspieszenie.
- Wydajność jądra istotnie zależy od konfiguracji wykonania:

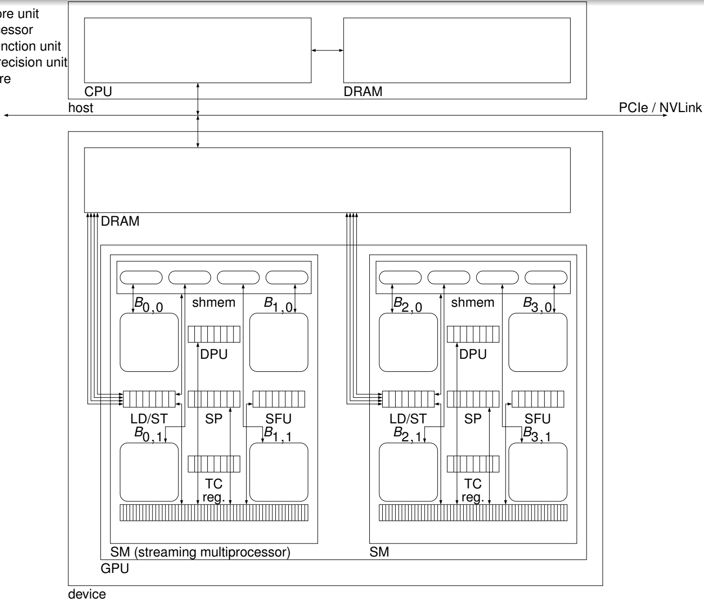
czas [ms]	$T_{B,x} = T_{B,y}$					
$B_{G,x} = B_{G,y}$	1	2	4	8	16	32
1				478.1	120.9	49.93
2			488.1	121.4	31.82	12.35
4		482.3	120.9	30.26	7.743	3.145
8	477.1	122.8	30.02	7.870	2.131	1.620
16	136.4	32.12	8.237	2.102	1.212	1.062
32	42.39	10.08	2.299	1.219	1.042	**1.025**

Analiza wydajności implementacji GPU

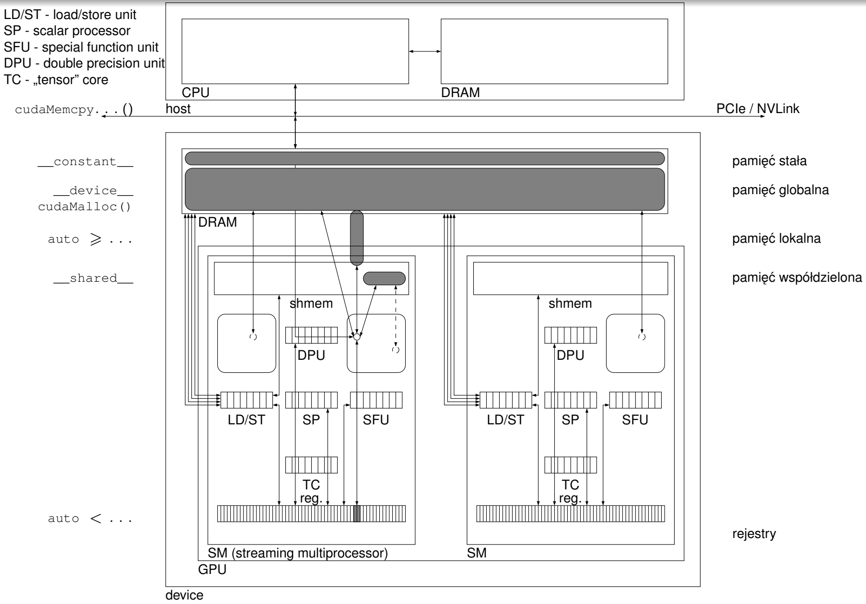
- Na RTX 4070 osiągnęliśmy wydajność ok. 2075.8 GFLOP/s, czyli przyspieszenie $S_u \approx 558$.
- Teoretyczna wydajność GPU wynosi ok. 35.48 TFLOP/s (2 FLOP $\cdot$ liczba rdzeni $\cdot$ częstotliwość zegara) – 17 $\times$ $>$!
- Ograniczeniem są m.in. dostępy do pamięci. Algorytm bazuje na obliczaniu iloczynu skalarnego (CGMA tylko $\frac{2p-1}{2p+1} \approx 1$), co oznacza, że dane są niepotrzebnie wielokrotnie czytane z pamięci.
- Przepustowość GPU wynosi ok. 126 GT/s. Po pomnożeniu przez CGMA dostaniemy 126 GFLOP/s. Nasza 16,5 $\times$ lepsza wydajność wynika z b. dużej pamięci *cache*.
- Sytuację można spróbować poprawić używając zamiast DDR DRAM dużo szybszej pamięci współdzielonej (SM),
- ...jest ona jednak znacznie mniejsza, więc trzeba będzie przetwarzać dane małymi porcjami kopiowanymi do niej.
- Kluczowe jest wykorzystanie hierarchii pamięci CUDA.

Hierarchiczny model sprzętowy systemu pamięci

LD/ST - load/store unit
SP - scalar processor
SFU - special function unit
DPU - double precision unit
TC - „tensor” core



Hierarchiczny model programowy systemu pamięci



Hierarchiczny model programowy systemu pamięci

Pamięć*	Alokacja	Dostępność: Miejsce	Dostępność: Czas	Gdzie jest?
globalna	`__device__ v[DIM]` `cudaMalloc(&v, DIM)`	sieć	aplikacja	DRAM
stała†	`__constant__ v[DIM]`	sieć	aplikacja	DRAM
współdzielona	`__shared__ v[DIM]`	blok	jądro	SM
lokalna	`auto v[A_LOT]`	wątek	jądro	DRAM SM$^{2.0}$ (L1)
rejestry	`auto v[A_FEW]`	wątek	jądro	SM

Istnieją funkcje `__isGlobal()`, `__isConstant()`, `__isShared()`...

*Pomijamy pamięć tekstur i pamięć powierzchni$^{2.0}$, gdyż używane są one przede wszystkim do grafiki, a nie obliczeń. Jednak w niektórych aplikacjach można wykorzystać filtracyjne właściwości tekstur np. do interpolacji.

†Jest to mała (64 KiB) pamięć, buforowana (*cached*) na GPU. Dla $C_c \geqslant 2.0$ (gdzie GPU jest wyposażony w cache L2) jest rzadziej używana.

Rozmiary poziomów hierarchii systemu pamięci [KiB]

Pamięć	Parametr	*2.0* *2.1*	*3.0*	*3.2* 3.5	5.0 5.3	6.1 7.0 8.6 8.9	6.0 6.2 7.5	8.0
globalna		∞ (pamięć zewnętrzna, rzędu 10 GiB)						
stała	C_{max}	64						
współdz.	$S_{M\ \text{max}}$	32×1,5			32×2	32×3	32×2	32×5
lokalna	$A_{T\ \text{max}}$	512						
rejestry (32 b)	$R_{M\ \text{max}}$ [Ki]	32	64					
	$R_{T\ \text{max}}$	63		255				

- Szybkie pamięci są małe, więc trzeba przetwarzać dane małymi porcjami, jawnie kopiowanymi do nich.
- Pamięć w rejestrach bywa większa niż w wolniejszej od nich pamięci współdzielonej – rejestry trzeba jak najlepiej wykorzystywać.

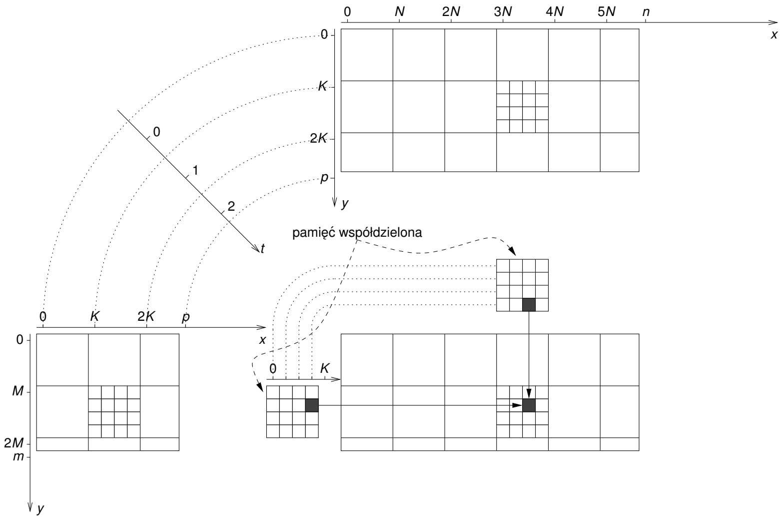
Mnożenie macierzy raz jeszcze – technika „kafelkowa”

- Przetwarzanie danych porcjami oznacza, że na „kafelki” musimy podzielić nie tylko macierz ***C***, ale także ***A*** i ***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” ***C***.
- Obliczenie elementów „kafelka” ***C*** przebiega etapami, podczas których skokowo poruszamy się po „kafelkach” ***A*** (w prawo blokowego wiersza) i ***B*** (w dół blokowej kolumny):

Mnożenie macierzy techniką „kafelkową” – algorytm

```
for all „kafelek” macierzy C do
    wyzeruj akumulator każdego elementu „kafelka” C
    for all „kafelek” wiersza A i kolumny B do
        pobierz kolejne kafelki A i B do pamięci współdzielonej
        poczekaj, aż oba „kafelki” zostaną całkowicie pobrane
        dodaj do akumulatora każdego elementu „kafelka” C
        cząstkowe iloczyny skalarne „kafelków” A i B
        poczekaj, aż wszystkie cząstkowe iloczyny skalarne
        pobranych „kafelków” A i B zostaną dodane
    end for
    zapisz akumulator każdego elementu „kafelka” C
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 spowoduje zakleszczenie!