Programowanie równoległe
Sprawozdanie z realizacji projektu
"procesory graficzne" - Wersja 1

Bartosz Adamczewski 151764
bartosz.adamczewski@student.put.poznan.pl
Ivan Kaliadzich 153936
ivan.kaliadzich@student.put.poznan.pl
Grupa L5 środa 8:00
Wymagany termin oddania - 31.05.2024
Termin oddania – 31.05.2024

1. Opis karty graficznej

Karta graficzna: NVIDIA GeForce RTX 2060

Układ scalony:
Nazwa modelu karty: NVIDIA GeForce RTX 2060
Układ scalony: TU106

Technologia:
Nazwa technologii: Turing
Parametr CC (Compute Capability):
Możliwości obliczeniowe: 7.5

Jednostki wykonawcze:
Liczba jednostek wykonawczych SM (Streaming Multiprocessors): 30
Liczba rdzeni CUDA: 1920
Jednostki zmiennoprzecinkowe SP (Single Precision): 1920
Jednostki zmiennoprzecinkowe DP (Double Precision): 1/32 wartości SP (czyli około 60)
Jednostki całkowitoliczbowe: 1920 (te same rdzenie co dla SP)
SFU (Special Function Units): 240 (8 na każdy SM)

Pamięć:
Rodzaj pamięci: GDDR6
Wielkość pamięci: 6 GB
Cache L2: 3 MB
Przepustowość pamięci: 336 GB/s

2. Sposób generowania tablicy wejściowej

```
50  // Generowanie losowych liczb
51  void MatrixRandom(float* matrix, int N) {
52      // Ziarno dla generatora liczb losowych
53      std::random_device rd;
54      std::mt19937 gen(rd());
55      std::uniform_real_distribution<> dis(0.0, 100.0);
56
57      for (int i = 0; i < N; i++) {
58          for (int j = 0; j < N; j++) {
59              matrix[i * N + j] = dis(gen);
60          }
61      }
62  }
```

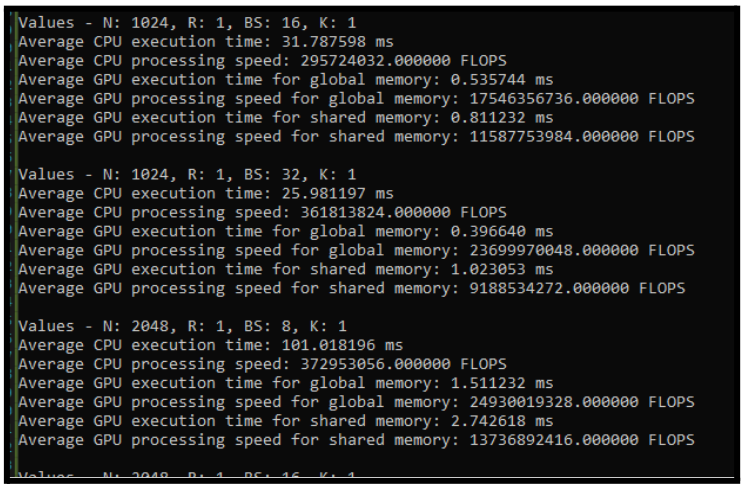
Funkcja wypełnia macierz o rozmiarze N x N losowymi liczbami zmiennoprzecinkowymi za pomocą generatora liczb losowych wykorzystującego losowo wygenerowane ziarno w przedziale od 0,0 do 100,0.

3. Poprawność obliczeń

```
33  // Sprawdzenie że 2 macierzy są takie same
34  void verifyMatrices(float* matrix1, float* matrix2, int N, int R) {
35      int matrix_size = N - 2 * R;
36      for (int i = 0; i < matrix_size * matrix_size; i++) {
37          if ((matrix1[i] != matrix2[i])) {
38              fprintf(stderr, "Discrepancy at index [%d]: %f != %f\n", i, matrix2[i], matrix1[i]);
39              exit(1);
40          }
41      }
42  }
```

Poprawność obliczeń weryfikujemy poprzez porównanie wszystkich elementów tablicy wyjściowej obliczonej przez CPU oraz przez GPU. Jeżeli wystąpi błąd zostanie on wypisany. Jeżeli nie ma błędu zostaną po prostu podane wyniki czasowe.

4. Przykładowy poprawny wynik wykonania programu:



```
Values - N: 1024, R: 1, BS: 16, K: 1
Average CPU execution time: 31.787598 ms
Average CPU processing speed: 295724032.000000 FLOPS
Average GPU execution time for global memory: 0.535744 ms
Average GPU processing speed for global memory: 17546356736.000000 FLOPS
Average GPU execution time for shared memory: 0.811232 ms
Average GPU processing speed for shared memory: 11587753984.000000 FLOPS

Values - N: 1024, R: 1, BS: 32, K: 1
Average CPU execution time: 25.981197 ms
Average CPU processing speed: 361813824.000000 FLOPS
Average GPU execution time for global memory: 0.396640 ms
Average GPU processing speed for global memory: 23699970048.000000 FLOPS
Average GPU execution time for shared memory: 1.023053 ms
Average GPU processing speed for shared memory: 9188534272.000000 FLOPS

Values - N: 2048, R: 1, BS: 8, K: 1
Average CPU execution time: 101.018196 ms
Average CPU processing speed: 372953056.000000 FLOPS
Average GPU execution time for global memory: 1.511232 ms
Average GPU processing speed for global memory: 24930019328.000000 FLOPS
Average GPU execution time for shared memory: 2.742618 ms
Average GPU processing speed for shared memory: 13736892416.000000 FLOPS
```

5. Kluczowe fragmenty kodu

a) Funkcja sekwencyjna wykonywana przez CPU - oblicza sumę elementów w podmacierzy o promieniu R dla każdego elementu macierzy wejściowej i zapisuje wyniki do macierzy wyjściowej. Kluczowe elementy kodu:

- linia 45 -> argumenty funkcji to rozmiar macierzy wejściowej, promień podmacierzy, wskaźniki na tablice wejściową oraz wyjściową.
- linie 46 - 47 -> pętle, które powodują zmniejszenie macierzy wyjściowej o R z każdej strony. Jest to spowodowane tym, że podmacierze skrajnych indeksów macierzy wejściowej, której elementy są sumowane, wychodziłaby poza macierz wejściową.
- linie 49 - 50 -> pętle, które sumują wszystkie elementy podmacierzy o określonym promieniu do zmiennej sum.

```
44  // Obliczanie tablicy sekwencyjnie (CPU)
45  void SumMatrixCPU(int N, int R, float* input_matrix, float* output_matrix) {
46      for (int i = R; i < N - R; i++) {
47          for (int j = R; j < N - R; j++) {
48              float sum = 0;
49              for (int ii = i - R; ii <= i + R; ii++) {
50                  for (int jj = j - R; jj <= j + R; jj++) {
51                      sum += input_matrix[ii * N + jj];
52                  }
53              }
54              output_matrix[(i - R) * (N - 2 * R) + j - R] = sum;
55          }
56      }
57  }
```

b) Funkcja równoległa wykonywana przez GPU z wykorzystaniem pamięci globalnej - Funkcja jądra CUDA (kernel), która jest uruchamiana na GPU i oblicza sumę elementów w podmacierzy o promieniu R dla k kolejnych elementów w wierszu w macierzy wejściowej i zapisuje wyniki do macierzy wyjściowej korzystając z pamięci globalnej GPU. Kluczowe elementy kodu:

- linia 60 -> argumenty funkcji to wskaźniki na tablicę wejściową oraz wyjściową, rozmiar macierzy wejściowej, promień podmacierzy oraz liczba wyników obliczanych przez jeden wątek.
- linia 62 - 63 -> obliczenie bazowych współrzędnych na podstawie indeksu bloku w siatce (blockIdx), liczby wątków w bloku (blockDim) oraz indeksu wątku w bloku. ‘j’ jest dodatkowo skalowane poprzez K.
- linia 65 -> pętla określająca liczbę indeksów macierzy wejściowej dla których ma zostać dokonane sumowanie
- linia 67 -> sprawdzenie czy indeksy znajdują się w zakresie macierzy
- linia 68 - 73 -> pętle sumują elementy w promieniu ‘R’ wokół elementu centralnego przydzielonego do wątku

```
59  // Jądro do przetwarzania macierzy z wykorzystaniem pamięci globalnej na GPU
60  __global__ void kernelGlobalMemory(float* input, float* output, int N, int R, int K) {
61      int matrix_size = N - 2 * R;
62      int i = threadIdx.x + blockIdx.x * blockDim.x;
63      int j = (threadIdx.y + blockIdx.y * blockDim.y) * K;
64
65      for (int k = 0; k < K; k++) {
66          float sum = 0;
67          if (i < matrix_size && j + k < matrix_size) {
68              for (int ii = -R; ii <= R; ii++) {
69                  for (int jj = -R; jj <= R; jj++) {
70                      sum += input[(j + k + R + ii) * N + (i + R + jj)];
71                  }
72              }
73              output[(j + k) * matrix_size + i] = sum;
74          }
75      }
76  }
```

c) Funkcja równoległa wykonywana przez GPU również obliczająca sumę elementów w podmacierzy o promieniu R, ale korzysta z pamięci współdzielonej bloku wątków. Kluczowe elementy kodu:

- Linia 117 -> argumenty funkcji to wskaźniki na tablicę wejściową oraz wyjściową, rozmiar macierzy wejściowej N, promień podmacierzy R, oraz liczba wyników obliczanych przez jeden wątek K.
- linia 121 - 123 -> obliczenie obszaru, który trzeba załadować do pamięci współdzielonej
- Linia 136 - 141-> pętla po K, w której każdy wątek ładuje dane z pamięci globalnej do pamięci współdzielonej. Każdy wątek ładuje część danych, a następnie wszystkie wątki są synchronizowane za pomocą __syncthreads(), aby upewnić się, że wszystkie dane są załadowane.
- Linia 143 - 151-> Obliczenia na danych w pamięci współdzielonej. Każdy wątek sumuje elementy w promieniu R wokół elementu centralnego przydzielonego do wątku, a następnie zapisuje wynik do pamięci globalnej.

```
116  // Jądro do przetwarzania macierzy z wykorzystaniem pamięci współdzielonej na GPU
117  __global__ void kernelSharedMemory(float* input, float* output, int N, int R, int K) {
118      extern __shared__ float sharedMemoii[];
119      unsigned int i = ((blockIdx.x * blockDim.x) + threadIdx.x) + R;
120      unsigned int j = ((blockIdx.y * blockDim.y * K) + threadIdx.y) + R;
121      unsigned int width = (N - 2 * R - blockIdx.x * blockDim.x >= blockDim.x) ? blockDim.x + 2 * R : N - 2 * R - blockIdx.x * blockDim.x + 2 * R;
122      unsigned int height = (N - 2 * R - blockIdx.y * blockDim.x >= blockDim.x) ? blockDim.x + 2 * R : N - 2 * R - blockIdx.y * blockDim.x + 2 * R;
123      unsigned int size = height * width;
124      unsigned int threadNumber = threadIdx.x * blockDim.x + threadIdx.y;
125      unsigned int threadX = threadIdx.x;
126      unsigned int threadY = threadIdx.y;
127
128      if (N < blockDim.x + 2 * R) {
129          width = N;
130      }
131
132      if (N < blockDim.x + 2 * R) {
133          height = N;
134      }
135
136      for (int k = 0; k < K; k++) {
137          for (unsigned int idx = threadNumber; idx < size; idx += blockDim.x * blockDim.x) {
138              int translatedIndex = getTranslatedIndex(idx, width, blockIdx.x * blockDim.x, blockIdx.y * blockDim.y * K + k * blockDim.x, N);
139              sharedMemoii[idx] = input[translatedIndex];
140          }
141          __syncthreads();
142
143          if (i < N - R && j < N - R) {
144              float total = 0;
145              for (unsigned int x = threadX; x <= threadX + 2 * R; x++)
146                  for (unsigned int y = threadY; y <= threadY + 2 * R; y++)
147                      total += sharedMemoii[y * width + x];
148
149              output[(i - R) * (N - 2 * R) + (j - R)] = total;
150              j += blockDim.x;
151          }
152
153          __syncthreads();
154      }
155  }
156
```

`

d) Funkcja SumMatrixGPU jest funkcją, która zapewnia potrzebne dane kernelom oraz uruchamia je korzystając zarówno z pamięci globalnej, jak i współdzielonej GPU, w zależności od wartości argumentu mode. Kluczowe elementy kodu:
   - Linia 246 - 247 -> alokacja pamięci globalnej na GPU
   - Linia 249 -> pętla wykonuje pięć powtórzeń, aby zmierzyć czas wykonania i uśrednić wyniki.
   - Linia 250 - 251 -> ‘cudaEventCreate’ tworzy zdarzenia startEvent i stopEvent do pomiaru czasu wykonania.
   - Linia 253 -> rozpoczęcie pomiaru czasu
   - Linia 255 - 259 -> sprawdzenie warunku, który decyduje o typie wykorzystywanej pamięci
   - Linia 261 -> zakończenie pomiaru czasu
   - Linia 262 -> przesłanie wartości macierzy wyjściowej z GPU na Hosta
   - Linia 273 -> weryfikacja wyników obliczonych przez CPU oraz GPU
   - Linia 276 - 279 -> obliczenie czasów oraz prędkości

```
240 void SumMatrixGPU(int N, int R, int K, int matrix_size, const char* mode, float* input_matrix, float* output_matrix, dim3 threadGrid, dim3 blockGrid, int
241     float* input, * output;
242     cudaEvent_t startEvent, stopEvent;
243     auto* hostOutput = (float*)malloc(matrix_size * matrix_size * sizeof(float));
244     float totalElapsedTime = 0.0f;
245 
246     cudaMalloc((void**)&input, N * N * sizeof(float));
247     cudaMalloc((void**)&output, matrix_size * matrix_size * sizeof(float));
248 
249     for (int iter = 0; iter < 5; iter++) {
250         cudaEventCreate(&startEvent);
251         cudaEventCreate(&stopEvent);
252         cudaMemcpyAsync(input, input_matrix, N * N * sizeof(float), cudaMemcpyHostToDevice);
253         cudaEventRecord(startEvent, nullptr);
254 
255         if (strcmp(mode, "global") == 0) {
256             kernelGlobalMemoii<<<blockGrid, threadGrid>>>(input, output, N, R, K);
257         } else if (strcmp(mode, "shared") == 0) {
258             kernelSharedMemoii<<<blockGrid, threadGrid, shared_memoii_size>>>(input, output, N, R, K);
259         }
260 
261         cudaEventRecord(stopEvent, nullptr);
262         cudaMemcpyAsync(hostOutput, output, matrix_size * matrix_size * sizeof(float), cudaMemcpyDeviceToHost);
263         cudaEventSynchronize(stopEvent);
264         float elapsedTime = 0;
265         cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
266         totalElapsedTime += elapsedTime;
267         cudaEventDestroy(startEvent);
268         cudaEventDestroy(stopEvent);
269     }
270 
271     cudaFree(input);
272     cudaFree(output);
273     verifyMatrices(output_matrix, hostOutput, N, R);
274     free(hostOutput);
275 
276     avg_time = totalElapsedTime / 5.0f;
277     float processedElements = (N - 2 * R) * (N - 2 * R);
278     float flops_per_element = (2 * R + 1) * (2 * R + 1); // liczba operacji dla każdego elementu wyjściowego
279     avg_speed = processedElements * flops_per_element / (avg_time / 1000.0f); // FLOPS
280 }
```

6. Wzory wykorzystane do obliczeń:

a) Definicje:
   - FLOPS: Jest to miara wydajności komputera, używana przy obliczeniach zmiennoprzecinkowych. Skrót ten oznacza Floating Point Operations Per Second (Operacje Zmiennoprzecinkowe na Sekundę).
   - Prędkość przetwarzania: Mierzy, ile operacji zmiennoprzecinkowych system może wykonać na sekundę.

b) Formuła obliczania prędkości:
   - Obliczanie przetworzonych elementów:
     $PE = (N - 2 \cdot R)^2$,
     $gdzie\ PE - liczba\ przetworzonych\ elementów,\ N - długość\ wierszu,\ R - promień$

     Formuła reprezentuje liczbę elementów w macierzy wyjściowej po zastosowaniu promienia *R*.

   - Obliczanie operacji na element:
     $OE - (2 \cdot R + 1)^2$,
     $gdzie\ OE - liczba\ operacji\ na\ element,\ R - promień$

     Każdy element w macierzy wyjściowej jest obliczany przez sumowanie elementów w kwadracie o boku *2R+1*.

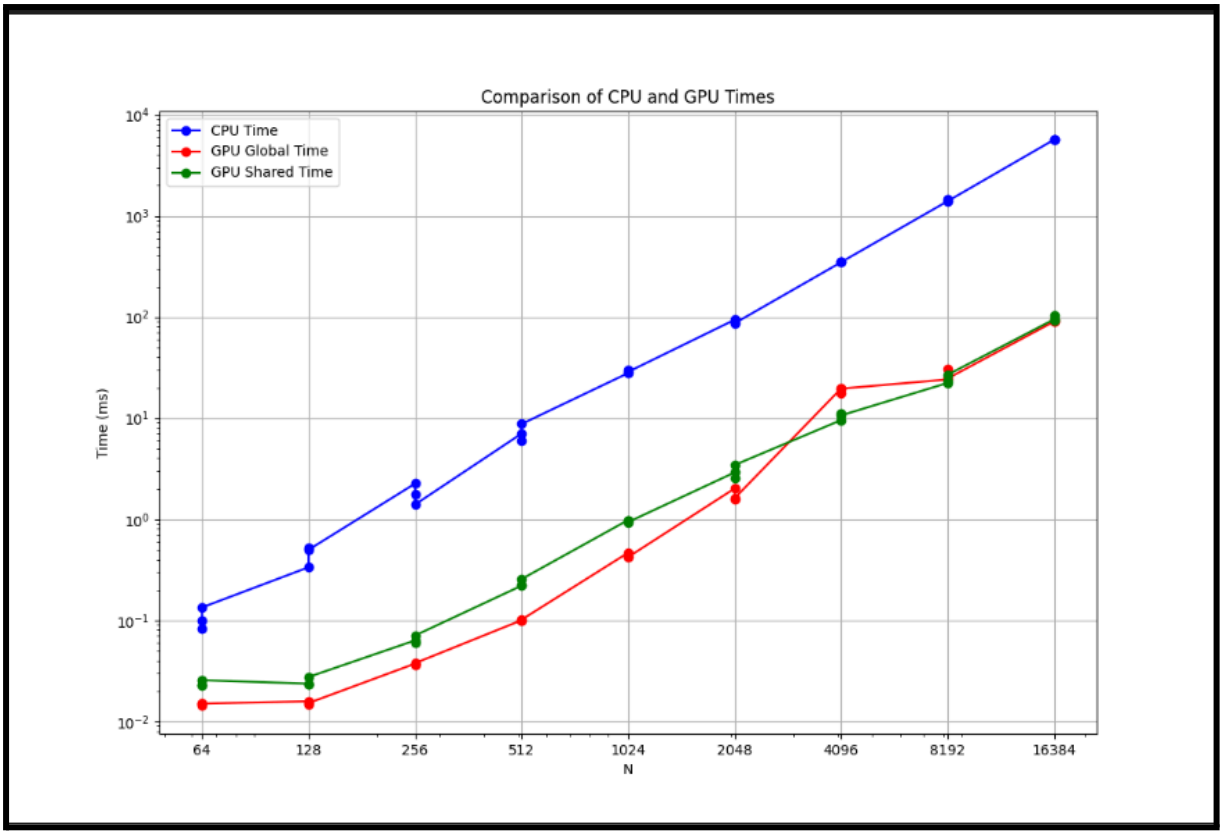
   - Obliczanie całkowitej operacji:
     $TO = PE \cdot OE$,
     $gdzie\ TO - łączna\ liczba\ operacji$

   - Obliczanie prędkości:
     $V\ [FLOPS] = \frac{TO}{T\ /\ 1000}$,
     $gdzie\ V - prędkość\ przetwarzania,$
     $T - czas\ przetwarzania\ (jest\ mierzony\ w\ milisekundach)$

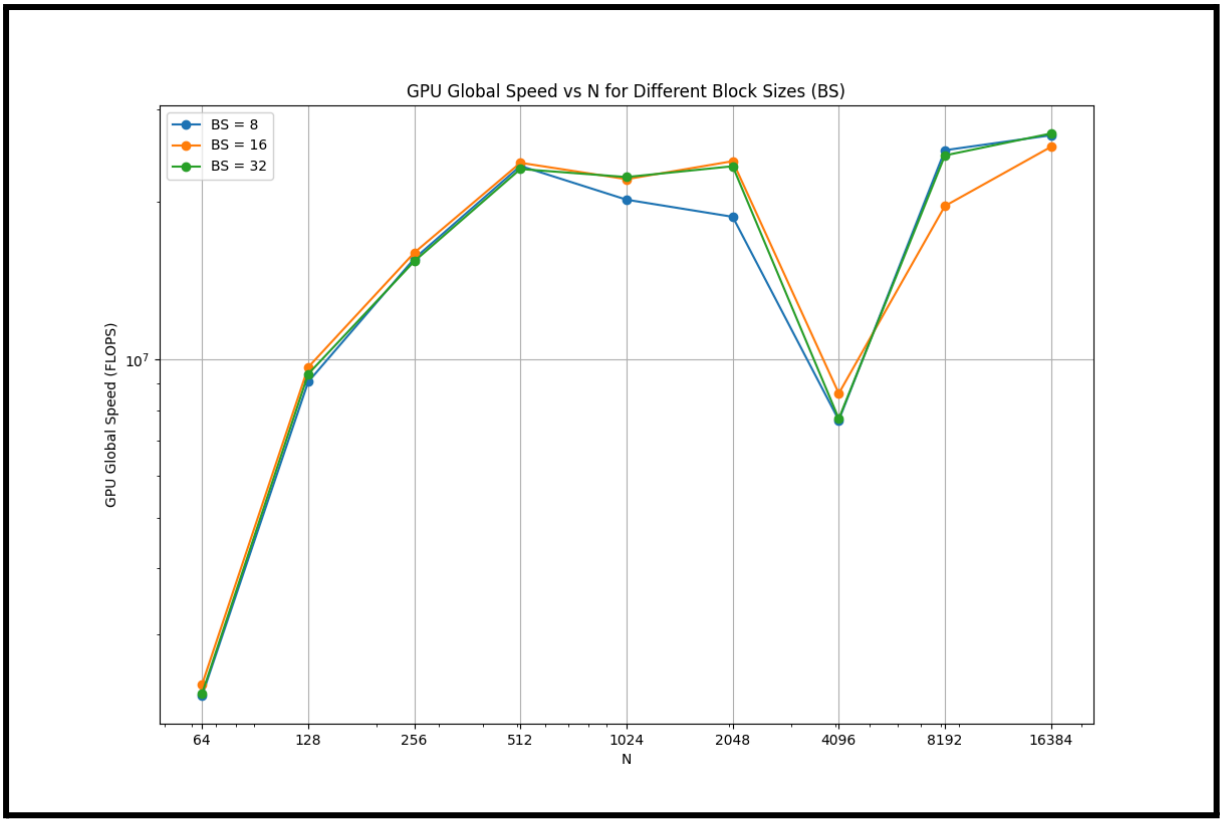
7. Wyniki eksperymentów

Czas [ms], prędkość [10^6 * flop/s]

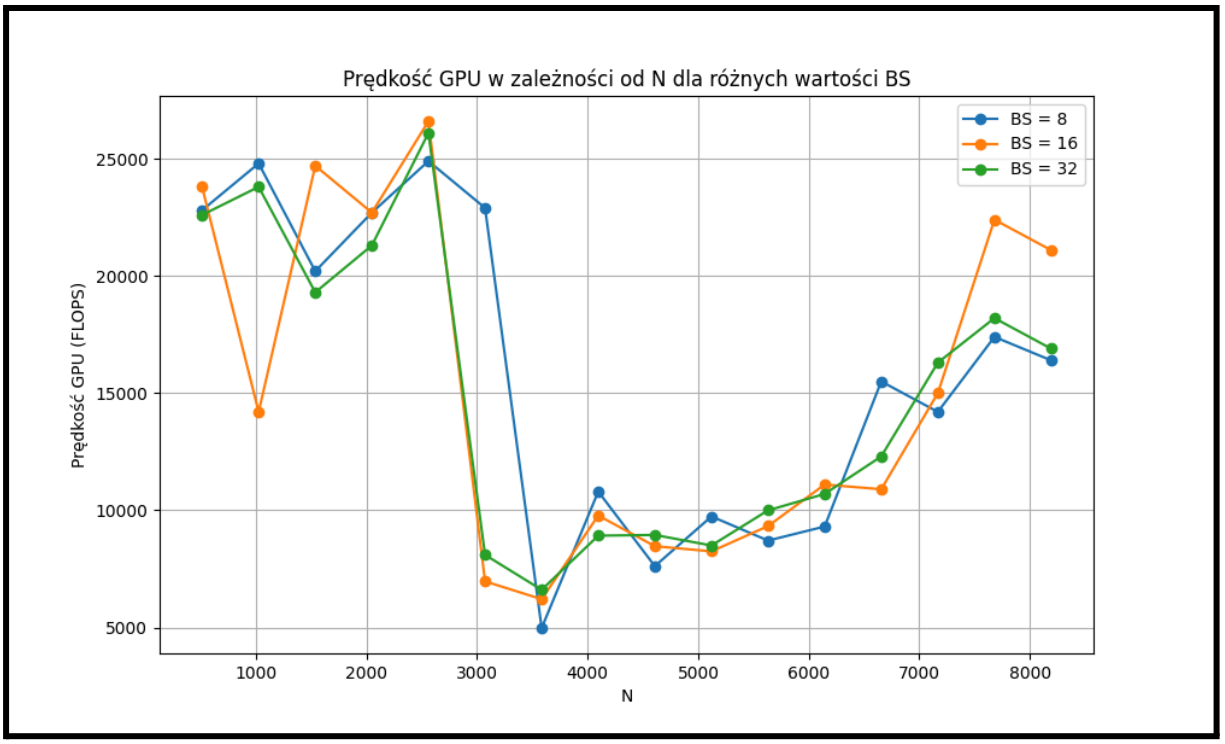
N	R	BS	K	CPU_time	CPU_speed	GPU_global_time	GPU_global_speed	GPU_shared_time	GPU_shared_speed
64	1	8	1	0,0998	346,653	0,0151	2292,462	0,0229	1508,687
64	1	16	1	0,0822	420,876	0,0144	2406,779	0,0230	1503,233
64	1	32	1	0,1342	257,794	0,0150	2312,072	0,0256	1352,082
128	1	8	1	0,3348	426,774	0,0158	9068,085	0,0236	6042,118
128	1	16	1	0,5208	274,355	0,0148	9668,958	0,0236	6058,514
128	1	32	1	0,4988	286,456	0,0152	9384,458	0,0275	5192,006
256	1	8	1	2,2526	257,766	0,0373	15567,197	0,0633	9167,910
256	1	16	1	1,7730	327,492	0,0363	15978,446	0,0598	9703,274
256	1	32	1	1,3976	415,458	0,0377	15387,657	0,0708	8206,749
512	1	8	1	7,0376	332,628	0,1002	23367,123	0,2202	10630,558
512	1	16	1	6,0052	389,812	0,0988	23691,016	0,2225	10518,969
512	1	32	1	8,7526	267,452	0,1015	23053,425	0,2559	9148,715
1024	1	8	1	27,8768	337,211	0,4664	20154,161	0,9815	9577,127
1024	1	16	1	30,0488	312,836	0,4263	22050,281	0,9802	9590,196
1024	1	32	1	28,6740	327,836	04224	22257,327	0,9366	10036,734
2048	1	8	1	93,9116	401,176	2,0150	18697,454	2,9012	12986,121
2048	1	16	1	88,1886	427,210	1,5790	23859,798	2,5259	14915,225
2048	1	32	1	86,7128	434,481	1,6144	23337,281	3,4380	10958,409
4096	1	8	1	348,3864	432,989	19,6768	7666,259	9,5587	15781,111
4096	1	16	1	348,1556	433,276	17,5020	8618,875	11,0429	13660,137
4096	1	32	1	348,9772	432,256	19,5525	7715,007	10,6399	14177,509
8192	1	8	1	1399,9460	431,220	24,1131	25035,512	22,2738	27102,872
8192	1	16	1	1452,0261	415,754	30,7896	19606,780	23,3369	25868,247
8192	1	32	1	1400,8251	430,950	24,6680	24472,396	26,8764	22461,555
16384	1	8	1	5684,6377	424,887	90,3533	26732,031	94,9476	25438,538
16384	1	16	1	5692,5747	424,295	95,0529	25410,364	92,2197	26191,034
16384	1	32	1	5685,2002	424,845	89,6472	26942,634	102,8085	23493,489



Wykres w skali logarytmicznej przedstawiający porównanie czasów wykonania. Pomimo bardzo małych wartości ‘R’ or ‘K’ prędkość z wykorzystaniem pamięci wspólnej bloku jest praktycznie taka sama co prędkość z użyciem pamięci globalnej co ukazuje potencjał tego podejścia przy obliczeniach, które zapewniają większą reużywalność danych. Czas CPU jest o rzędy większy od pozostałych dwóch metod co ukazuje przewagę GPu nad CPU w tego typu obliczeniach.



Wykres w skali logarytmicznej przedstawiający zmianę prędkości obliczeń dla rosnącego rozmiaru macierzy wejściowej wykorzystując pamięć globalną.

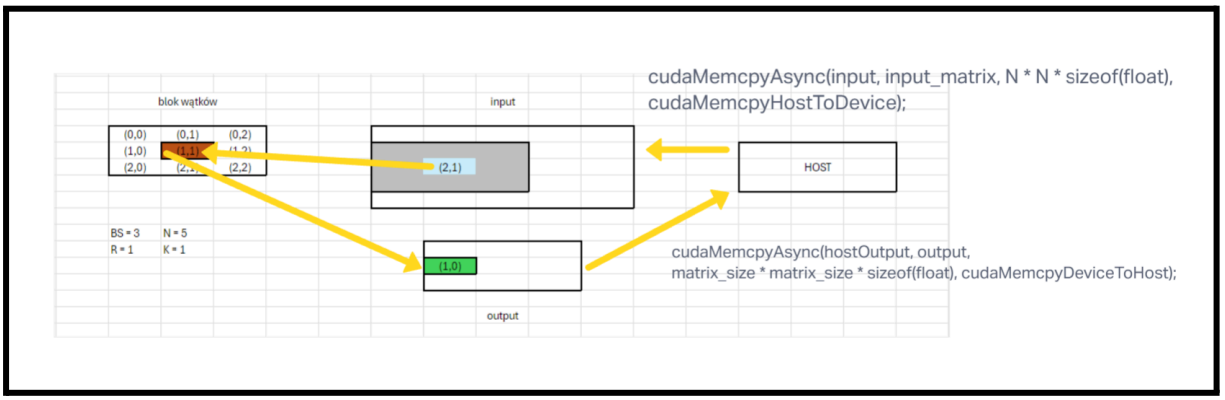


Wykres dla ograniczonej dziedziny aby lepiej określić punkt nasycenia.

Punkt nasycenia w kontekście obliczeń równoległych, szczególnie na GPU, odnosi się do momentu, w którym dalsze zwiększanie rozmiaru problemu lub liczby zasobów (np. wątków, bloków) przestaje prowadzić do proporcjonalnego wzrostu wydajności obliczeń. WIdzimy na tym wykresie, że po przekroczeniu N=1024 nie widać istotnego wzrostu prędkości obliczeń. Po przekroczeniu N = 2500 zauważalny jest nawet duży spadek prędkości obliczania. Głównymi czynnikami wpływającymi na punkt nasycenia są: przepustowość pamięci oraz zasoby obliczeniowe. Do dalszych obliczeń wybraliśmy wartość 2048, która jest istotnie większa od punktu nasycenia. Ciężko na powyższych wykresach zobaczyć zależność pomiędzy prędkością oraz rozmiarem bloku wątków. Na obydwu wykresach wartości są do siebie zbliżone i często następuje zmiana na ‘prowadzeniu’. Jeżeli trzeba byłoby określić faworyta, to byłby nim BS = 32. Najwidoczniej ten rozmiar bloku najlepiej wykorzystuje zasoby GPU, umożliwiając najbardziej optymalne prowadzenie dużej ilości obliczeń.

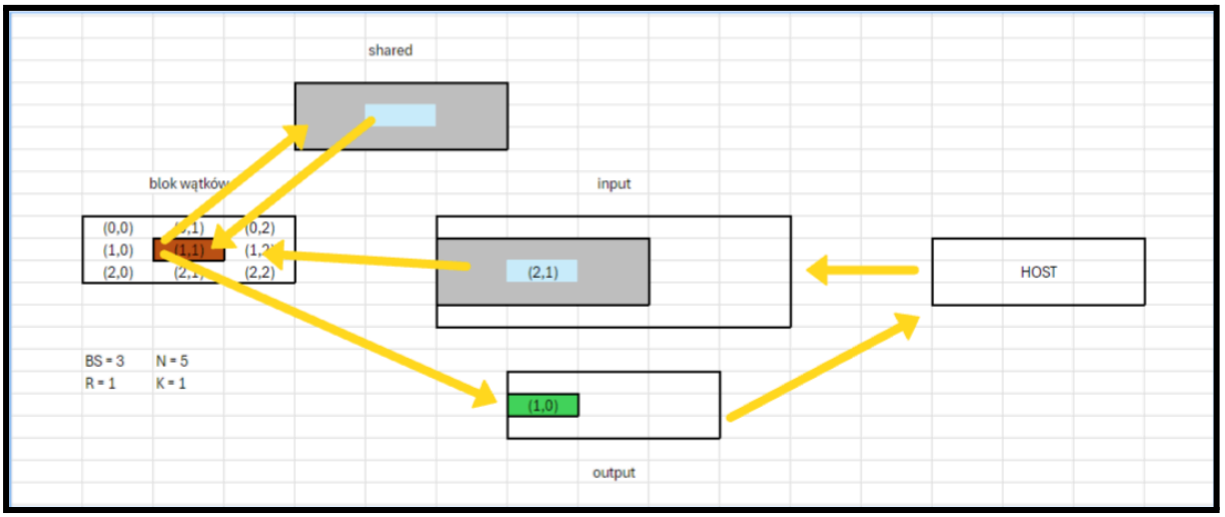
8. Rysunki ukazujące działanie programu

a) Obliczanie z wykorzystaniem pamięci globalnej



Wykonanie obliczeń na GPU zaczynamy od zaalokowania pamięci globalnej za pomocą ‘cudaMalloc’. Następnie za pomocą ‘cudaMemcpy’ przekazujemy zawartość wcześniej wypełnionej macierzy wejściowej do pamięci globalnej GPU. Po tym funkcja ‘kernelGlobalMemory’ oblicza na podstawie współrzędnych wątku w bloku (1,1), współrzędnych bloku w gridze (0,0) oraz na podstawie rozmiaru bloku (3) pozycje wątku w globalnej macierzy wątków. ‘j’ mnożone jest przez K, ponieważ zwiększa to lokalność danych. Wynika to z zamienienia struktury 2D na strukturę 1D. W dalszej części wątek sumuje lokalnie odpowiadający mu indeks pobierając dane z pamięci globalnej do pamięci lokalnej wątku (rejestry). Na koniec następuje zapisanie wyniku do pamięci globalnej. Problem polega na tym, że przy większym rozmiarze ‘R’ oraz ‘K’ wiele razy te same dane pobierane są z wolniejszej pamięci globalnej.

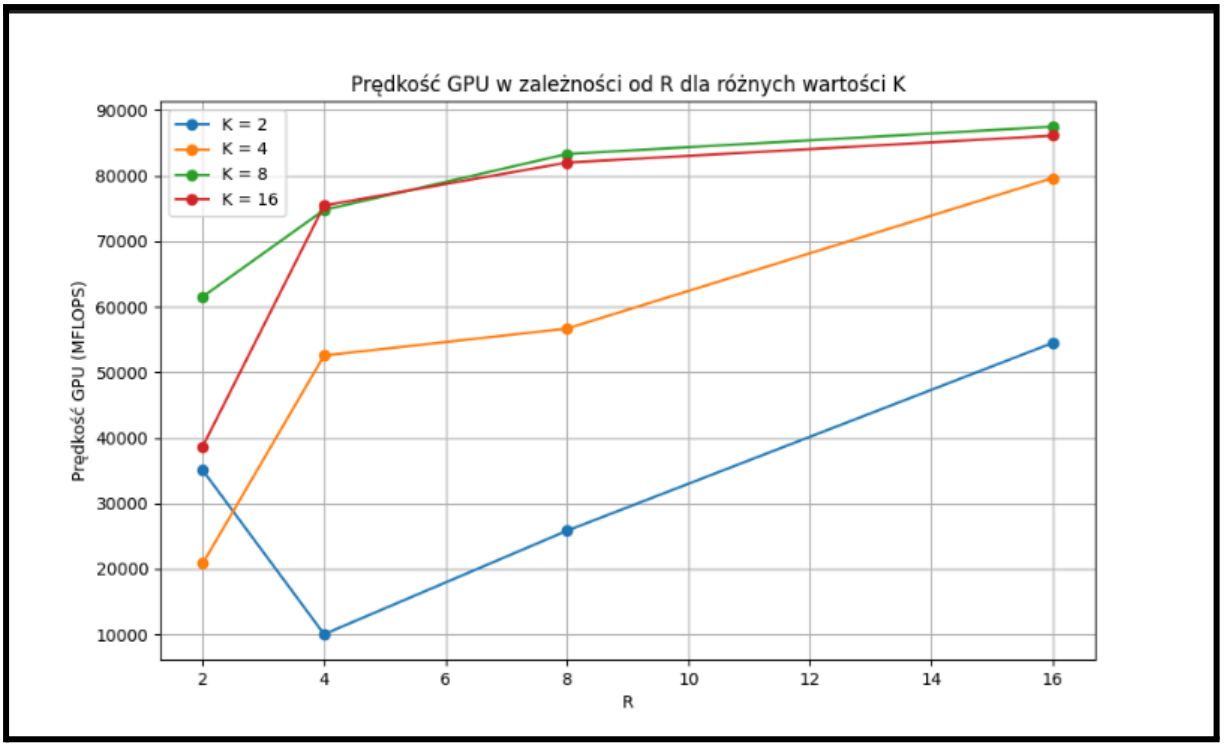
b) Obliczanie z wykorzystaniem pamięci współdzielonej

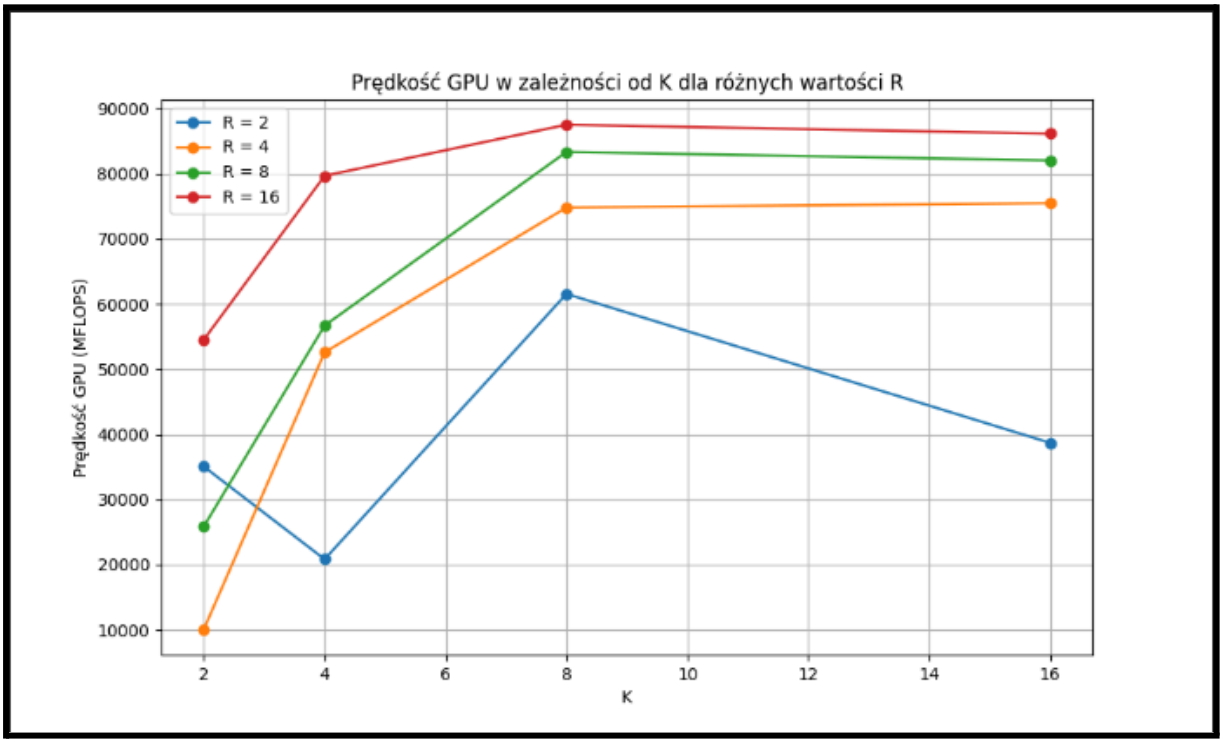


Główna różnica polega na tym co dzieje się po załadowaniu macierzy wejściowej do pamięci globalnej GPU. Wszystkie wątki w bloku przenoszą potrzebne im do obliczeń wartości z pamięci globalnej do pamięci wspólnej bloku. W tak małym przypadku operacja ta może wydawać się mało opłacalna, lecz przy większej wartości ‘R’ oraz ‘K’ jest to jak najbardziej optymalne, ponieważ pamięć współdzielona jest o rzędy szybsza od pamięci globalnej. Na plus wpływa także typ zadania wykonywanego przez GPU, ponieważ raz załadowane dane do pamięci współdzielonej będą wiele razy używane. Aby zachować poprawność obliczeń potrzebne jest przeliczanie indeksów jednej macierzy na indeksy drugiej. Istotna jest synchronizacja wątków pomiędzy ładowaniem danych do pamięci współdzielonej, a wykonywaniem obliczeń za pomocą ‘syncthreads’, aby mieć pewność, że wszystkie dane są gotowe do wykonywania obliczeń.

Tabela z pomiarami, które ukazują wpływ ‘K’, ‘R’ oraz ‘BS’ na prędkość i czas przetwarzania. Rozmiar macierzy obliczaliśmy zgodnie ze wzorem: N(k,R) = (N_nas-2)*k+2*R. Jako N_nas przyjęliśmy wcześniej określoną wartość 2048.

N	R	BS	K	CPU Time (ms)	CPU Speed (MFLOPS)	GPU Time (ms)	GPU Speed (MFLOPS)
4096	2	32	2	916,838013	456,581856	11,938304	35064,57805
8188	2	32	4	3640,123047	459,997184	80,393631	20828,09856
16372	2	32	8	15175,68945	441,349664	108,833374	61541,65043
32740	2	32	16	61740,125	433,934048	692,572632	38683,50874
4100	4	32	2	2983,85498	454,546752	135,104919	10038,87616
8192	4	32	4	13437,23438	403,744288	103,189636	52575,10502
16376	4	32	8	49047,07031	442,44896	290,157654	74789,77331
32744	4	32	16	195301,9219	444,45696	1150,289551	75462,13171
4108	8	32	2	10922,67578	443,037024	187,318253	25833,84269
8200	8	32	4	43607,91406	443,878144	341,486633	56683,33158
16384	8	32	8	192656,3594	401,88864	929,242798	83322,03622
32752	8	32	16	694745,0625	445,783104	3776,299805	82012,9792
4124	16	32	2	41621,83203	438,104736	334,622009	54493,4953
8216	16	32	4	166762,625	437,381536	915,673279	79656,02611
16400	16	32	8	668333,125	436,542112	3334,450195	87497,35117
32768	16	32	16	265800,097	439,258615	13550,69272	86122,69655

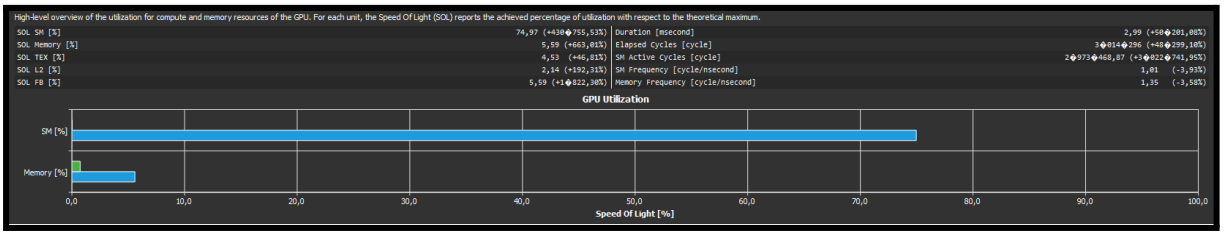


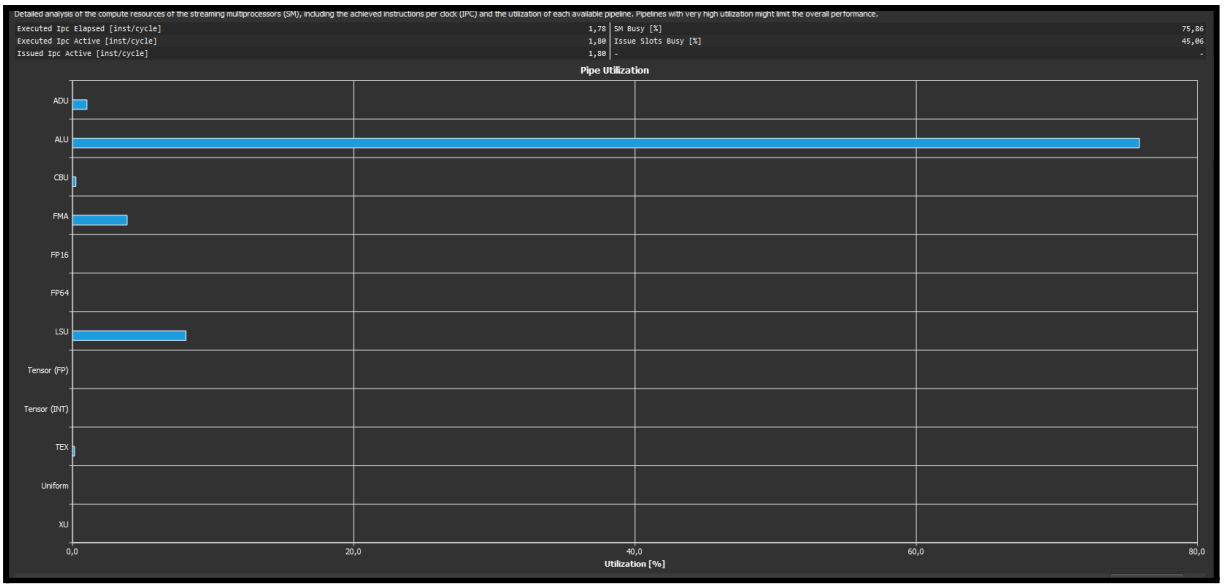


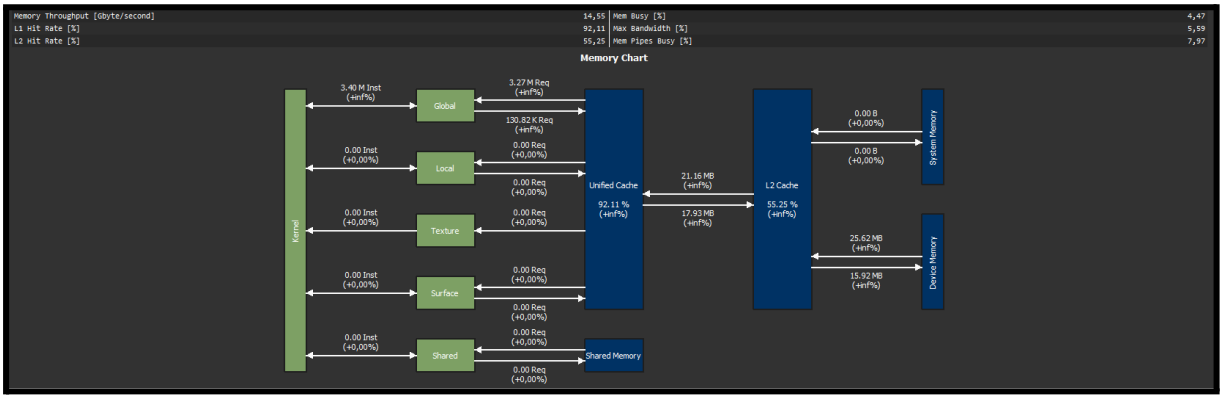
Wzrost liczby wątków ‘K’ znacząco poprawia prędkość GPU dla małych rozmiarów problemów ‘R’. Dla dużych rozmiarów problemów ‘R’, prędkość GPU stabilizuje się przy wyższych wartościach K, wskazując na osiągnięcie optymalnego poziomu równoległości. Wartości K powyżej 8 nie przynoszą znaczących korzyści w kontekście prędkości GPU dla dużych wartości R, sugerując punkt nasycenia. Ogólnie, większe wartości R pozwalają na lepsze wykorzystanie możliwości równoległego przetwarzania GPU, jednakże efektywność przestaje rosnąć po przekroczeniu pewnego pułapu.

9. Wyniki analizy programu NVIDIA Nsight

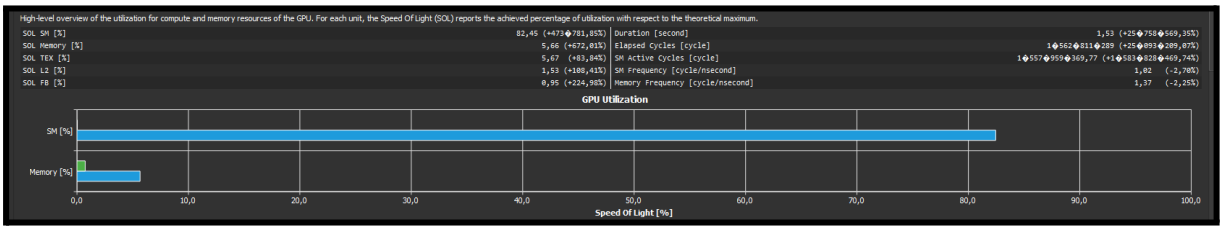
   a) Wynik dla wykonania ‘kernelGlobalMemory’ z parametrami N = 1024, R = 2, BS = 32 oraz K = 2.

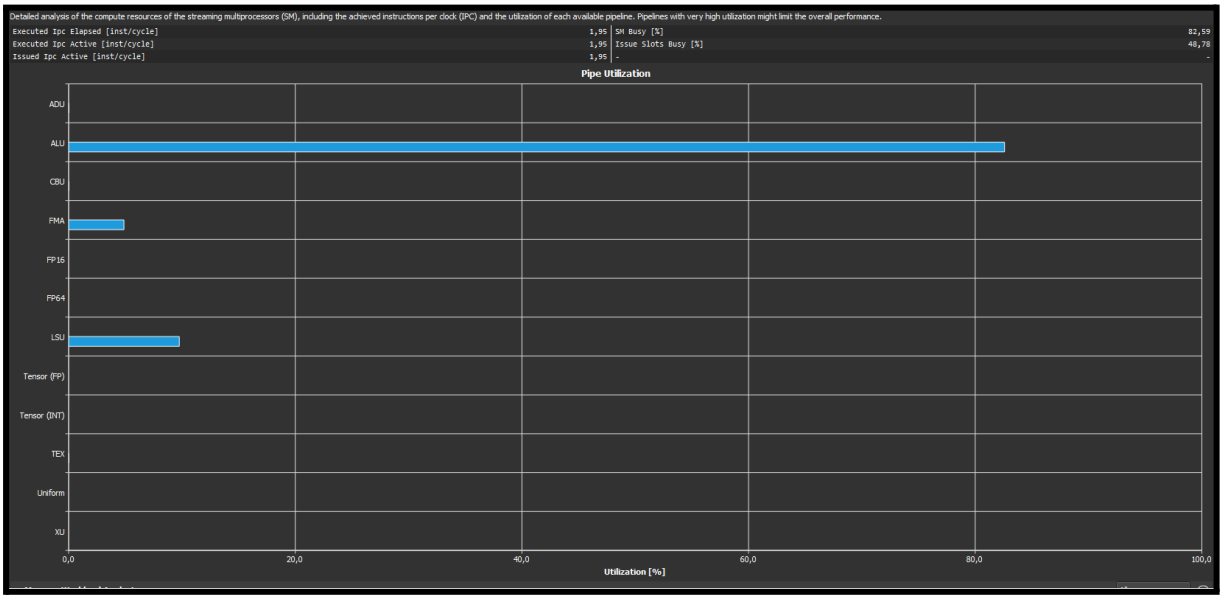


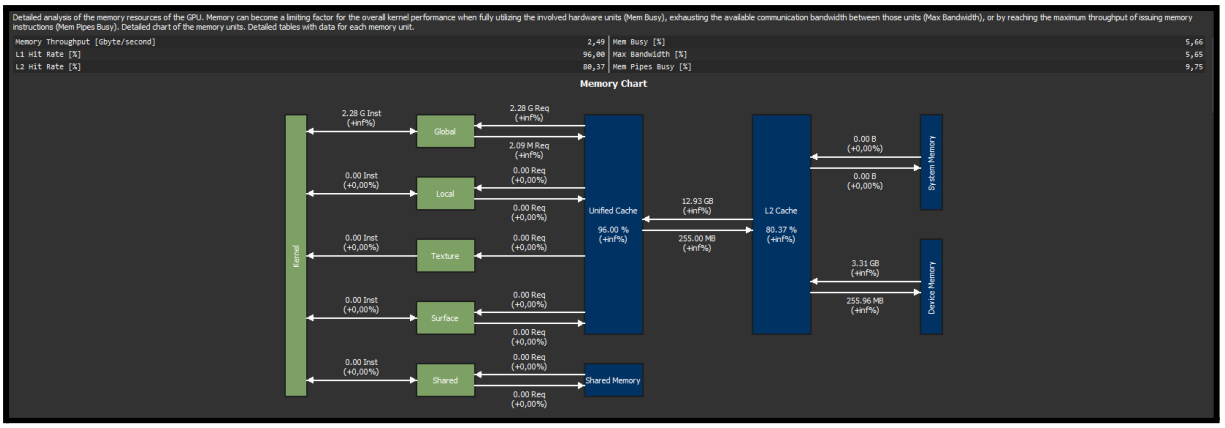




b) Wynik dla wykonania ‘kernelGlobalMemory’ z parametrami N = 1024, R = 16, BS = 32 oraz K = 8.







W przypadku drugiego zestawu parametrów (R = 16, K = 8), wykorzystanie SM wzrosło do 82,45%, w porównaniu do 74,97% w pierwszym przypadku (R = 2, K = 2). Oznacza to bardziej efektywne wykorzystanie jednostek obliczeniowych GPU przy większych wartościach R i K. Skutkiem tego jest to, że czas trwania operacji w drugim przypadku jest krótszy (1,53 sekundy) w porównaniu do pierwszego przypadku (2,99 sekundy). W obu przypadkach wykorzystanie pamięci pozostaje na niskim poziomie (~5,59% i 5,66%), co wskazuje, że ograniczenia pamięciowe nie są głównym wąskim gardłem w tych scenariuszach. Wzrost wartości R i K prowadzi do lepszego wykorzystania zasobów obliczeniowych GPU (SM), co skutkuje wyższą efektywnością i krótszym czasem przetwarzania. Niskie wykorzystanie pamięci sugeruje, że optymalizacja obliczeń w

jednostkach SM ma kluczowe znaczenie dla poprawy wydajności w tych scenariuszach. Wzrost wartości R i K prowadzi do znacznego wzrostu obciążenia pamięci globalnej i cache L2, co skutkuje wyższą efektywnością i większą przepustowością pamięci. Wysokie wykorzystanie Unified Cache sugeruje, że optymalizacja dostępu do tej pamięci ma kluczowe znaczenie dla poprawy wydajności w tych scenariuszach.

10. Podsumowanie

Dlaczego GPU jest szybsze niż CPU:

- GPU jest szybsze niż CPU ze względu na samo przygotowanie architektury do obliczeń równoległych. Procesor ma stosunkowo mniej rdzeni, ale są one bardziej elastyczne, ale nie jest tak zoptymalizowany do obliczeń na szeroką skalę, jak karta graficzna. Oznacza to w praktyce, że tam, gdzie obliczenia mogą być rozbite na wiele mniejszych, niezależnych części, GPU przetwarza je znacznie szybciej.

Wpływ wartości K, R i BS na prędkość przetwarzania K (Liczba wyników obliczanych przez jeden wątek):

- Wartość K ma znaczący wpływ na prędkość przetwarzania. Większe wartości K zwiększają efektywność przetwarzania równoległego na GPU. Jednak po osiągnięciu pewnej wartości (około K=8 dla dużych problemów) dalszy wzrost K nie przynosi istotnych korzyści, wskazując na osiągnięcie punktu nasycenia. Oznacza to, że dla większych wartości K zasoby obliczeniowe GPU są już optymalnie wykorzystane.
- R (Promień podmacierzy): Większe wartości R zwiększają ilość danych, które muszą być przetworzone przez każdy wątek. Większy promień R powoduje lepsze wykorzystanie równoległych zdolności GPU, ponieważ zwiększa to lokalność danych i zmniejsza ilość operacji pamięciowych.
- BS (Rozmiar bloku wątków): Rozmiar bloku (BS) również wpływa na prędkość przetwarzania. Optymalny rozmiar bloku pozwala na najlepsze wykorzystanie zasobów GPU. W sprawozdaniu, rozmiar bloku BS=32 był często najlepszy, ponieważ pozwalał na najbardziej efektywne prowadzenie dużej ilości obliczeń. Optymalny rozmiar bloku zależy jednak od specyfiki zadania i struktury danych.

Wyniki eksperymentów pokazują, że wartości K, R i BS mają kluczowy wpływ na efektywność obliczeń na GPU. Przy odpowiednim doborze tych parametrów, GPU może przetwarzać dane znacznie szybciej niż CPU, co zostało udowodnione poprzez znacząco krótsze czasy przetwarzania i wyższe prędkości obliczeniowe (w MFLOPS) w przypadku wykorzystania GPU.