HPC CUDA 1.

Wojciech Raczuk 459487

Implementacja “worker”

Zdecydowałem się dać 32 thready w bloku - dzięki temu każdy thread może trzymać swój ~1024 (maksymalna liczba wierzchołków) wymiarowy wycinek tablicy booli not_visited[] w shared memory; taka tablica waży 32 * 1024 = 32KB. Dostęp do tej tablicy uzyskujemy poprzez makro obok - dzięki temu, że ustawiliśmy MAX_N na liczbę nieparzystą, wątki z jednego warpa, które próbują symultanicznie uzyskać dostęp do tego samego indeksu, nie będą powodowały bank conflictu.

```
#define cord(x, y) ((x) * n + (y))
#define tour_pos(i) (n * (i) + blockDim.x * blockIdx.x + threadIdx.x)
#define MAX_N (1031)
```

```
#define WORKER_THREAD_NUM       32
#define warp_pos(i)             (MAX_N * threadIdx.x + (i))
```

```
// Construct the tour
float prob[MAX_N + 4];
for (int step = 1; step < n; step++) {
    float sum_prob = 0.0;
    // Calculate the probabilities for each unvisited city
    for (int j = 0, cr = cord(current_city, 0); j < n; j++, cr++) {
        float tmp = prob_numerator[cr] * not_visited[warp_pos(j)];
        prob[j] = tmp;
        sum_prob += tmp;
    }

    float rand_val = curand_uniform(&states[idx]) * sum_prob;
    float cumulative_prob = 0.0;
```

Implementacja “worker”

Prawdopodobieństwa, wyrażone 4 bajtowym floatem, zajmują za dużo pamięci, by móc naiwnie zmieścić je w shared memory. Stąd też umieścimy je w VRAMie. Dostęp do tej pamięci kolejnym makrem nie jest najwydajniejszy, ale wobec względenej losowości current_city, ciężko o łączony (coalesced) memory access.

Zapis do tour[tour_pos(i)] również powinien się łączyć.

```
#define cord(x, y) ((x) * n + (y))
#define tour_pos(i) (n * (i) + blockDim.x * blockIdx.x + threadIdx.x)
#define MAX_N (1031)
```

```
#define WORKER_THREAD_NUM       32
#define warp_pos(i)             (MAX_N * threadIdx.x + (i))
```

```
// Construct the tour
float prob[MAX_N + 4];
for (int step = 1; step < n; step++) {
    float sum_prob = 0.0;
    // Calculate the probabilities for each unvisited city
    for (int j = 0, cr = cord(current_city, 0); j < n; j++, cr++) {
        float tmp = prob_numerator[cr] * not_visited[warp_pos(j)];
        prob[j] = tmp;
        sum_prob += tmp;
    }

    float rand_val = curand_uniform(&states[idx]) * sum_prob;
    float cumulative_prob = 0.0;
```

Implementacja “worker”

Warto też zwrócić uwagę, że na liczbach zmiennoprzecinkowych niekoniecznie (wartość sumy) * (losowa liczba z [0, 1]) jest nie większe niż wartość tej sumy ewaluowanej po kolei od lewej do prawej.

```
while (cumulative_prob < rand_val && j + 1 < n) {
    j++;
    cumulative_prob += prob[j];
}

// Nienawidzę floatów.
if (cumulative_prob < rand_val) {
    while (!not_visited[j]) {
        j--;
    }
}
```

Optymalizacja “worker”

Zauważmy, że dotychczasowy fragment kodu wykonuje największą pracę (o rząd wielkości większą niż odzysaknie wyniku, aktualizacja feromonów itd.). Jest tu największe pole do optymalizacji. Wymyśliłem, że można pogrupować prawdopodobieństwa np. po 16. Wtedy mamy 16x mniejsze zapotrzebowanie pamięciowe na tablicę prob[], co jest wystarczające żeby zmieścić się w 48KB! Potem w najwyżej 16 operacjach można sprawdzić, które prawdopodobieństwo wypada w ruletce.

Optymalizacja “worker”

Można pokusić się jeszcze o pogrupowanie prob_numeratorów w czwórki i wykonanie wektorowych odczytów oraz odrobinę ręcznej wektoryzacji.

Później trzeba *bardzo* uważać na ruletce

Przyniosło to zauważalne efekty na moim GTX 1080Ti.

```
int n4ceil = (n + 3) / 4;
for (int i = 0; i < n; i += 4) {
    prob_numerator4[n4ceil * idx + (i / 4)] = {
        prob_numerator[cord(idx, i)],
        prob_numerator[cord(idx, i + 1)] * (i + 1 < n),
        prob_numerator[cord(idx, i + 2)] * (i + 2 < n),
        prob_numerator[cord(idx, i + 3)] * (i + 3 < n)
    };
}
```

```
#define PROB_GROUP_SIZE 16
int n4ceil = (n + 3) / 4;
int n5ceil = (n + PROB_GROUP_SIZE - 1) / PROB_GROUP_SIZE;
for (int j = 0, pg = 0, cr = n4ceil * current_city; j < n; pg++) {
    float4 fetch = prob_numerator4[cr]; cr++;
    prob[prob_gr_pos(pg)]  = fetch.x * not_visited[warp_pos(j)] * (j < n); j++;
    prob[prob_gr_pos(pg)] += fetch.y * not_visited[warp_pos(j)] * (j < n); j++;
    prob[prob_gr_pos(pg)] += fetch.z * not_visited[warp_pos(j)] * (j < n); j++;
    prob[prob_gr_pos(pg)] += fetch.w * not_visited[warp_pos(j)] * (j < n); j++;

    if constexpr (PROB_GROUP_SIZE / 4 > 1) {
        fetch = prob_numerator4[cr]; cr++;
        prob[prob_gr_pos(pg)] += fetch.x * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.y * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.z * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.w * not_visited[warp_pos(j)] * (j < n); j++;
    }

    if constexpr (PROB_GROUP_SIZE / 4 > 2) {
        fetch = prob_numerator4[cr]; cr++;
        prob[prob_gr_pos(pg)] += fetch.x * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.y * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.z * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.w * not_visited[warp_pos(j)] * (j < n); j++;
    }

    if constexpr (PROB_GROUP_SIZE / 4 > 3) {
        fetch = prob_numerator4[cr]; cr++;
        prob[prob_gr_pos(pg)] += fetch.x * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.y * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.z * not_visited[warp_pos(j)] * (j < n); j++;
        prob[prob_gr_pos(pg)] += fetch.w * not_visited[warp_pos(j)] * (j < n); j++;
    }

    sum_prob += prob[prob_gr_pos(pg)];
}
```

Optymalizacja “worker”, ale…

Choć lokalnie wszystko zdawało się działać (nie mogę tego ponownie zweryfikować, ponieważ musiałem wyjechać), to na entropii pojawiają się dziwne błędy - co ciekawe, o różnym nasileniu w zależności od GPU.

Kod załączam w worker2.cuh.

Narzędzia Nvidii nie są zbyt pomocne. Co powinno się zrobić?

```
CUDA Exception: Warp Illegal Address
The exception was triggered at PC 0x100002fe928  worker_tour_construction(float*, int, float, float, int*, c

Thread 1 "main" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
[Switching focus to CUDA kernel 0, grid 7, block (17,0,0), thread (0,0,0), device 0, sm 17, warp 0, lane 0]
0x00000100002fe930 in worker_tour_construction<<<(32,1,1),(32,1,1)>>> (
    pheromones=0x7fffc3600000, n=1002, alpha=1, beta=2, tour=0x7fffc3200000,
    states=0x7fffcc400000) at /home/wraczuk/hpc1/worker.cuh:45
45                  float4 fetch = prob_numerator4[cr++];
p cr
(cuda-gdb) $1 = 135792
p &prob_numerator4
(cuda-gdb) $2 = (@global float4 (*)[1062961]) 0x7fffc2000000
p prob_numerator4[cr]
(cuda-gdb) $3 = {x = 3.30361114e-17, y = 2.66712576e-17, z = 4.67784566e-17,
  w = 4.51565904e-17}
```

```
CUDA Exception: Warp Out-of-range Address
The exception was triggered at PC 0x7fffd1a56890  worker_tour_construction(float*, int, float, float, int*,

Thread 1 "main" received signal CUDA_EXCEPTION_5, Warp Out-of-range Address.
[Switching focus to CUDA kernel 0, grid 7, block (31,0,0), thread (0,0,0), device 0, sm 62, warp 1, lane 0]
0x00007fffd1a568b0 in worker_tour_construction<<<(32,1,1),(32,1,1)>>> (
    pheromones=0x7fffbc000000, n=1002, alpha=1, beta=2, tour=0x7fffd5c00000,
    states=0x7fffcde00000) at /home/wraczuk/hpc1/worker.cuh:36
36          for (int step = 1; step < n; step++) {
p step
(cuda-gdb) $1 = 1001
p n
(cuda-gdb) $2 = 1002
```

Implementacja “queen”

W tej implementacji zgodnie podobnie jak w załączonej pracy, zdecydowałem się na 128 threadów.

Główną przewagą tej implementacji jest wygoda w użytku shared memory. Dostępy do prob[] i not_visited[] w obrębie warpa to kolejne indeksy (a nawet adresy), które leżą w innych bankach.

Podobnie jak wcześniej, wektoryzacja powoduje dziwne błędy.

```
#define QUEEN_THREAD_NUM        128
#define WARP_SIZE               32
#define q_tour_pos(i)           (n * block_id + (i))
```

```
__shared__ int current_city;
__shared__ bool not_visited[MAX_N];
__shared__ float prob[MAX_N];
```

```
for (int step = 1; step < n; step++) {
    float sum_prob = 0.;
    for (int j = idx, cr = cord(current_city, idx); j < n; j += blockDim.x, cr += blockDim.x) {
        float tmp = prob_numerator[cr] * not_visited[j];
        prob[j] = tmp;
        sum_prob += tmp;
    }
    atomicAdd(&sh_sum_prob, sum_prob);
    __syncthreads();
```

```
if (idx == 0) {
    sum_prob = sh_sum_prob;
    float rand_val = curand_uniform(&states[block_id]) * sum_prob;
    float cumulative_prob = 0.;
    int j = -1;

    while (cumulative_prob < rand_val && j + 1 < n) {
        j++;
        cumulative_prob += prob[j];
    }

    // Nienawidzę floatów.
    if (cumulative_prob < rand_val) {
        while (!not_visited[j]) {
            j--;
        }
    }

    current_city = j;
    tour[q_tour_pos(step)] = j;
    not_visited[j] = false;
}
__syncthreads();
```

Optymalizacja “queen”

Potencjalną optymalizacją jest pozbycie się atomica, na rzecz warpowej redukcji. W tym celu można użyć standardowej funkcji o dosyć samoopisowym syntaxie.

Okazuje się, że w praktyce takie podejście działa wolniej.

Uśredniając czas dla odpowiednio 100/10k iteracji na RTX 2080Ti:

Redukcja:
pr1002 807,887 ms
d198 16,886 ms

Atomic:
pr1002 748,92 ms
d198 12,4303 ms

```
for (int step = 1; step < n; step++) {
    float sum_prob = 0.;
    for (int j = idx, cr = cord(current_city, idx); j < n
        float tmp = prob_numerator[cr] * not_visited[j];
        prob[j] = tmp;
        sum_prob += tmp;
    }
    atomicAdd(&sh_sum_prob, sum_prob);
```

```
using WarpReduce = cub::WarpReduce<float>;
__shared__ typename WarpReduce::TempStorage temp_storage[4];
for (int step = 1; step < n; step++) {
    float sum_prob = 0.;
    for (int j = idx, cr = cord(current_city, idx); j < n; j += blockDim
        float tmp = prob_numerator[cr] * not_visited[j];
        prob[j] = tmp;
        sum_prob += tmp;
    }
    __syncwarp();
    int warp_id   = threadIdx.x / 32;
    float aggregate = WarpReduce(temp_storage[warp_id]).Sum(sum_prob);
    if (idx % WARP_SIZE == 0) {
        atomicAdd(&sh_sum_prob, aggregate);
    }
    __syncthreads();

    if (idx == 0) {
        sum_prob = sh_sum_prob;
        sh_sum_prob = 0.;
```

Reszta implementacji

Workflow jest identyczne w obu przypadkach. Trzeba synchronicznie zbudować ścieżki, przeliczyć feromony (korzystam po prostu z atomiców), a następnie pomocniczo wyliczamy i zapamiętujemy współczynniki.

Najintensywniej używaną pamięć po stronie hosta pinnujemy. Można też równoleglić kopię do len_h z kernelem calculate_prob_numerator, ale nie ma to wpływu na wydajność.

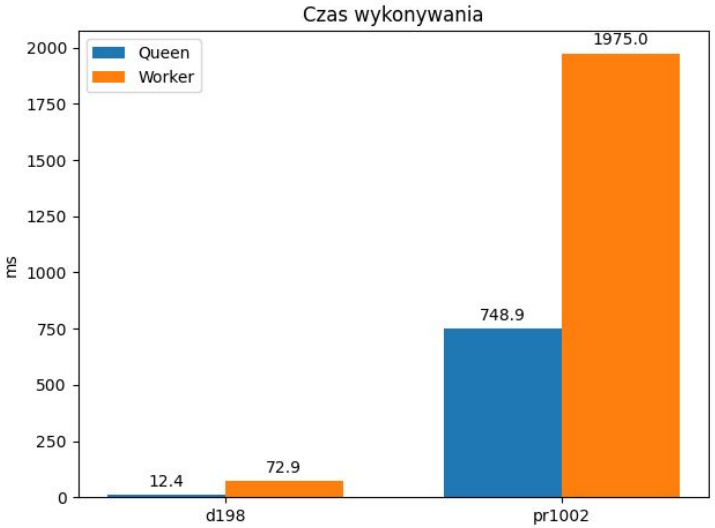
```
float min_len = 1e18;
int *tour_h;
float* len_h;
cudaHostAlloc((void**)&len_h, sizeof(float) * n, cudaHostAllocDefault);
cudaHostAlloc((void**)&tour_h, sizeof(int) * n * n, cudaHostAllocDefault);
int id = -1;
for (int i = 0; i < num_iter; i++) {
    queen_kernel<<<n, QUEEN_THREAD_NUM>>>(
            n, tour, states
    );

    queen_update_pheromones_and_lengths<<<(n + WARP_SIZE - 1) / WARP_SIZE, WARP_SIZE>>>(
            pheromones, n, tour, len, evaporate
    );

    calculate_prob_numerator<<<(n + WARP_SIZE - 1) / WARP_SIZE, WARP_SIZE>>>(
            pheromones, n, alpha
    );

    cudaMemcpy(len_h, len, sizeof(float) * n, cudaMemcpyDeviceToHost);

    if (min_len > *std::min_element(len_h, len_h + n)) {
        min_len = *std::min_element(len_h, len_h + n);
        id = std::min_element(len_h, len_h + n) - len_h;
        cudaMemcpy(tour_h, tour, sizeof(int) * n * n, cudaMemcpyDeviceToHost);
    }
```

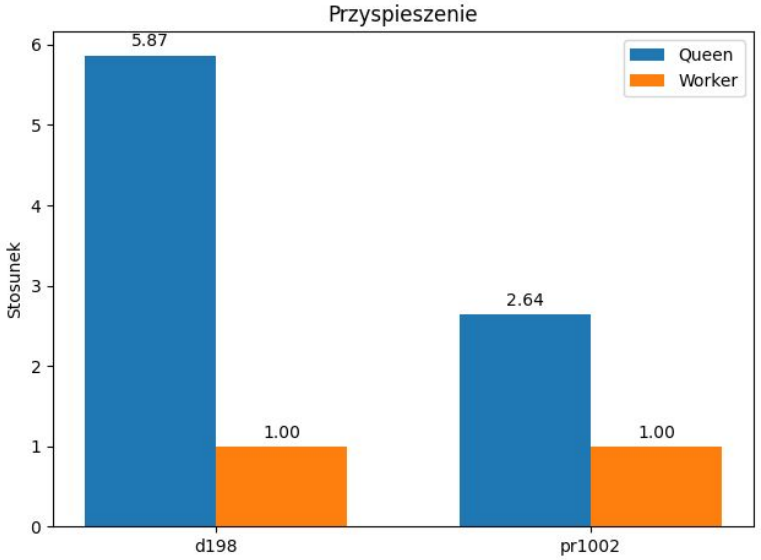
Wyniki

Implementacje przetestowałem na dwóch skrajnych testach - małym d198 i dużym pr1002. Na małym teście wykonanych było po 10k iteracji, a na dużym po 100. W obu przypadkach użyłem RTX 2080Ti na serwerze Entropy.

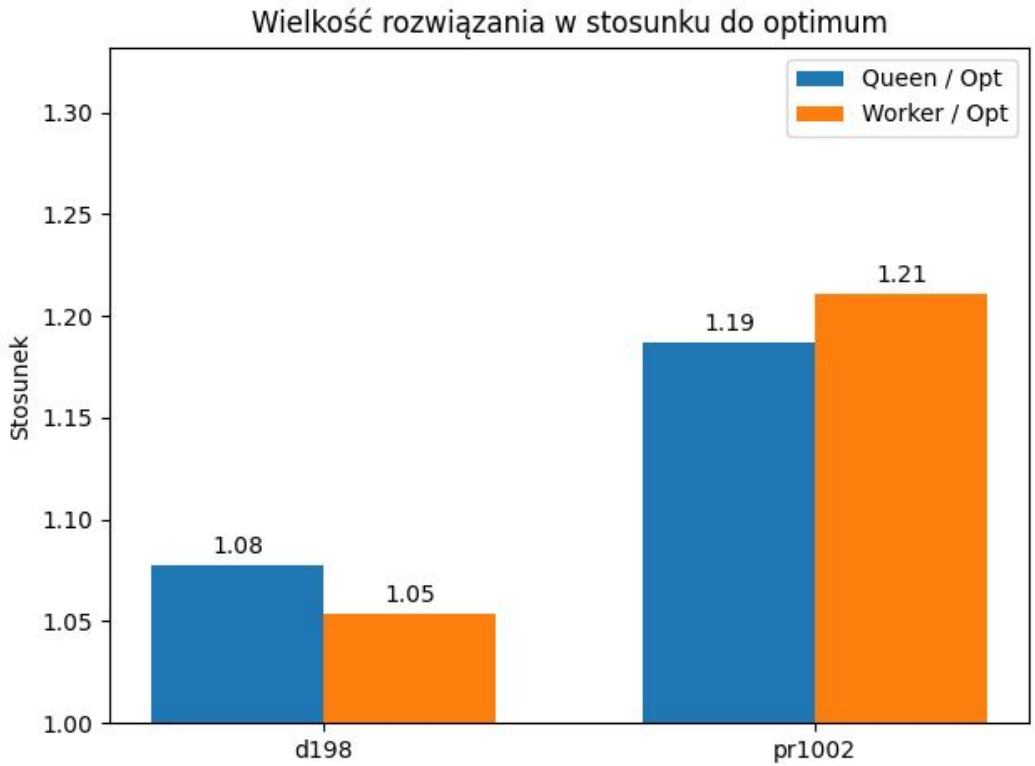
Czasy mierzyłem przy pomocy cudaEvents oraz nvprof.

Wyniki - czas wykonania





Wyniki - jakość rozwiązań



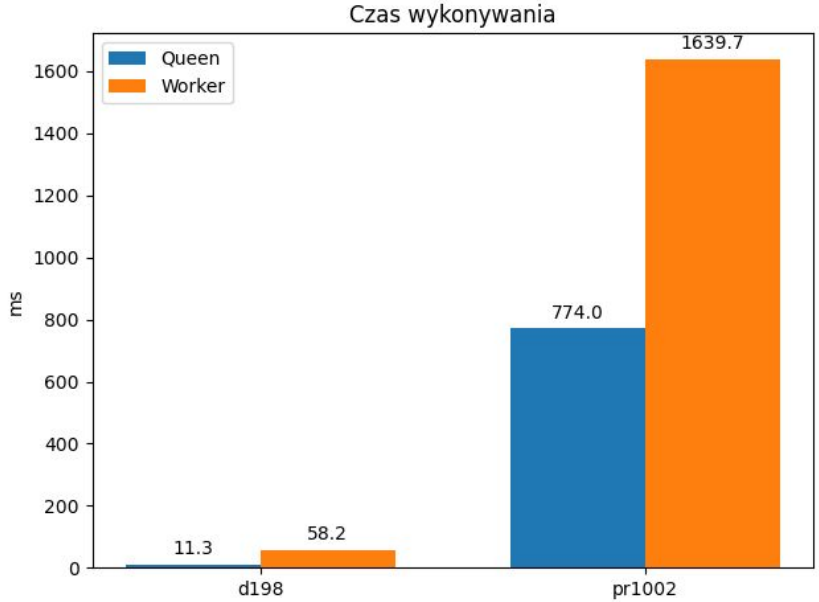
Wyniki (cudaGraph)

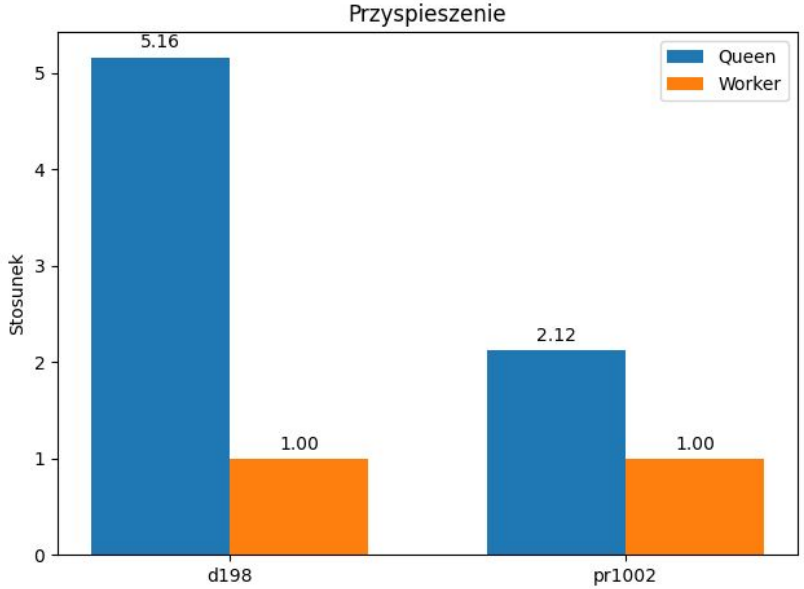
Kod łatwo przystosować do wykorzystania cudaGraph:
(odpowiedni define jest na początku maina)

```
#ifdef CUDAGRAPH
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    queen_kernel<<<n, QUEEN_THREAD_NUM, 0, stream>>>(
        n, tour, states
    );

    queen_update_pheromones_and_lengths<<<(n + WARP_SIZE - 1) / WARP_SIZE, WARP_SIZE, 0, stream>>>(
            pheromones, n, tour, len, evaporate
    );

    calculate_prob_numerator<<<(n + WARP_SIZE - 1) / WARP_SIZE, WARP_SIZE, 0, stream>>>(
            pheromones, n, alpha
    );

    cudaGraph_t graph;
    cudaStreamEndCapture(stream, &graph);

    cudaGraphExec_t instance;
    cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);
#endif
```

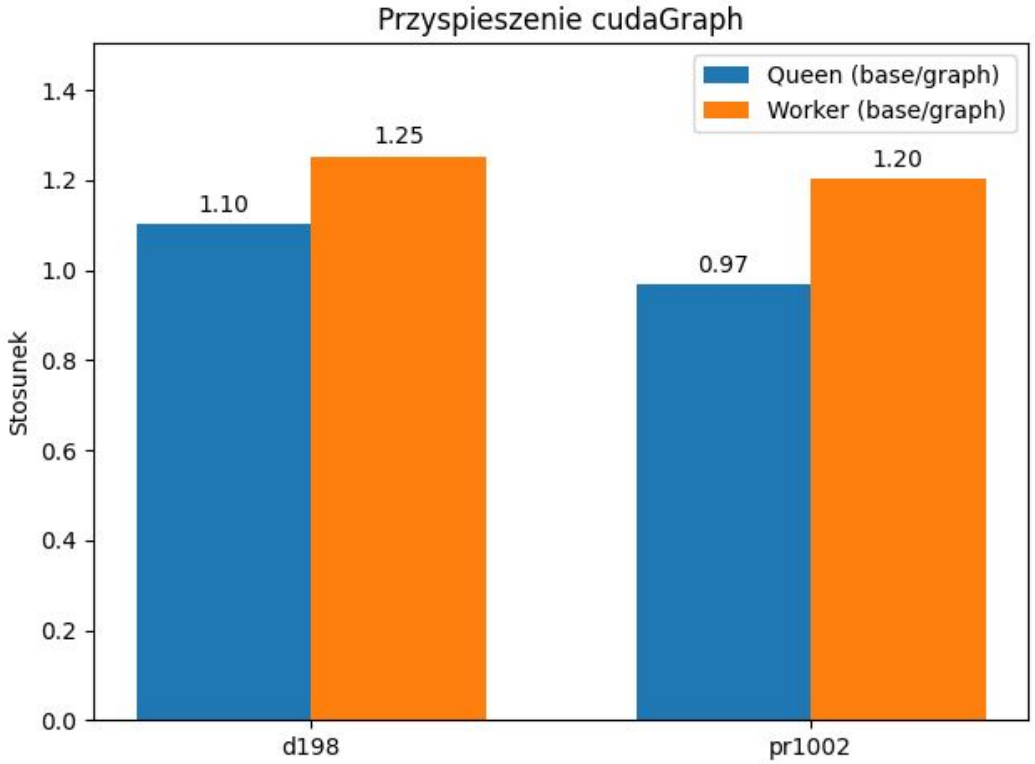
```
for (int i = 0; i < num_iter; i++) {
    #ifdef CUDAGRAPH
        cudaGraphLaunch(instance, stream);
        cudaStreamSynchronize(stream);
    #else
        queen_kernel<<<n, QUEEN_THREAD_NUM>>>(
                n, tour, states
        );

        queen_update_pheromones_and_lengths<<<(n + WARP_SIZE - 1) / WARP_SIZE, WARP_SIZE>>>(
                pheromones, n, tour, len, evaporate
        );

        calculate_prob_numerator<<<(n + WARP_SIZE - 1) / WARP_SIZE, WARP_SIZE>>>(
                pheromones, n, alpha
        );
    #endif
```

Wyniki (cudaGraph) - czas wykonania





Wyniki - przyspieszenie cudaGraph



Wnioski

Implementacja queen jest dużo wydajniejsza niż worker. Stosunki między implementacjami są zbliżone do tych z pracy.

Obie implementacje osiągają zadowalającą jakość aproksymacji - lepszą niż te przedstawione w pracy.

Cuda graph ma zauważalny wpływ na czas wykonania dla workera.
Dla queen różnica jest widoczna na małym teście - na dużym zmiana jest niewielka, choć spowalniająca.