# Algorytm SLAM z pominięciem czujników odometrycznych w kontekście różnych typów procesorów obliczeniowych

**Mateusz Fiedeń, Michał Miotk, Przemysław Dąbek, Artur Muraszkowski**

Wrocław University of Science and Technology,
Wydział Mechaniczny, ul. Łukasiewicza 7/9, 50-371 Wrocław,
*e-mail: mateusz.fieden@edu.pl*

***Streszczenie:*** *SLAM jest to algorytm równoczesnego mapowania otoczenia i lokalizowania się na tworzonej mapie. Wykorzystywany jest w robotach autonomicznych przeznaczonych do pracy w nieznanym bądź dynamicznie zmieniającym się otoczeniu. W swojej podstawowej formie wykorzystuje czujnik odległości, taki jak lidar bądź radar oraz dane o przesunięciu pozyskiwane z enkoderów. Dzięki zastosowaniu odpowiednich strategii dodawania kolejnych skanów oraz filtracji pobieranych danych uzyskuje się dokładne mapy, jednak użycie enkoderów, nie zawsze jest możliwe. W artykule poruszony zostaje temat pozycjonowania i mapowania przy użyciu lidaru bez wykorzystywania dodatkowych czujników zapewniających dane odometryczne. Zaproponowany zostaje odpowiedni algorytm oraz dyskusja dotycząca zastosowanych procesorów obliczeniowych, na których jest uruchamiany (wyłącznie CPU oraz z wykorzystaniem GPU wspierającego technologię CUDA). Zaprezentowane są wyniki w formie wykresów zależności czasu od iteracji, uzyskanych chmur punktów, a także parametrów sprzętowych obserwowanych w trakcie działania algorytmu.*

***Słowa kluczowe:*** *cyfrowe przetwarzanie obrazów, SLAM, CUDA, brak czujników odometrycznych*

# SLAM algorithm without odometric sensors usage in context of different computing processor types

***Abstract:*** *SLAM stands for a simultaneous localization and mapping. It's used in construction of autonomic robots, designed for work in topographically unknown areas or dynamically changing environment. In its simplest form it utilizes distance sensor, lidar for example, and displacement data obtained from encoders. Thanks to application of appropriate strategies of adding next scan iterations and filtration of obtained data, it allows to create accurate maps with minimal computing power required.*

*However, usage of encoders is not always possible, as in case of boats, legged robots or drones. To solve this problem, there's proposed an algorithm that allows for localization and mapping in described situation, with a discussion on type of processors used by program. Because of the task specifics, it's necessary to match many obtained simultaneously measurements with created map. For this purpose, the differences between algorithm version using only CPU, by spreading the task between different processor threads, and algorithm version that utilize graphical computing acceleration, that make calculations on many parallel CUDA cores, were checked.*

*Both implementations were tested on the corridor inside building with results in the form of charts comparing time needed for separated iterations to complete.*

***Keywords:*** *digital image processing, SLAM, CUDA, digital image processing, absence of odometric sensors*

# 1. Wstęp

Jednym z podstawowych problemów stojących na drodze zwiększenia autonomiczności robotów są ich ograniczone możliwości orientacji w przestrzeni. Z reguły obszar ich użyteczności ogranicza się do środowiska, w którym znane jest przynajmniej najbliższe otoczenie, a zatem nie zmienia się ono bądź też przewidywana jest jego kolejna iteracja. Wykonywanie czynności bez udziału człowieka w przestrzeni uprzednio nierozpoznanej jest znacząco utrudnione.

Większość poleceń podejmowanych przez robota wymaga precyzyjnego określenia położenia. Błędne szacunki prowadzić mogą do uszkodzeń wywołanych próbami przemieszczania się poza dostępną, wolną przestrzeń, doprowadzając do kolizji z obiektami. Konieczne jest zatem prawidłowe rozpoznanie oraz zapamiętanie geometrii miejsca, w obrębie którego robot ma zamiar pracować.

Tematyka zajmująca się zagadnieniem jednoczesnego określania własnej pozycji i zbierania informacji o otoczeniu, mająca pomóc w rozwiązaniu opisanego problemu, określana jest jako SLAM (Simultaneous Localisation and Mapping). W podstawowej formie metody SLAM skalują się kwadratowo względem liczby punktów orientacyjnych zbieranych z otoczenia [1]. Z tego powodu ich implementacja w systemach czasu rzeczywistego jest szczególnie wymagająca względem stosowanych rozwiązań. Istotny jest przede wszystkim czas z jakim dokonywane jest pobieranie oraz analiza danych. Osiągnięcie zadowalających rezultatów, poza zmniejszaniem złożoności stosowanych algorytmów, jest również powiązane ze sprzętem, na którym dokonywane są obliczenia.

Jednym z podstawowych czynników sprzętowych mających wpływ na czas przebiegu algorytmów SLAM, tak jak w większości zadań powiązanych z komputerowym przetwarzaniem obrazu, jest wybór pomiędzy CPU oraz GPU. CPU (centralna jednostka obliczeniowa) jest elementem odpowiedzialnym za pobieranie danych umieszczonych w pamięci operacyjnej celem ich interpretacji oraz ewentualnego wykonania powierzonych w ten sposób zadań. GPU (procesor graficzny) jest wyspecjalizowanym w kierunku przetwarzania obrazu, które z reguły bywa zbyt dużym obciążeniem dla podstawowego procesora, i w większości przypadków radzi sobie z nim zdecydowanie szybciej [2].

Zasadniczą różnicę pomiędzy CPU oraz GPU stanowi sposób ich działania, gdzie CPU zoptymalizowany jest pod kątem wykonywania pojedynczej czynności w obrębie zleconego zadania w możliwie najkrótszym czasie, podczas gdy GPU operuje na znacznej liczbie osobnych procesów jednocześnie. Przedmiotem badań były przede wszystkim różnice czasowe powstające przy realizacji algorytmu typu SLAM, przeprowadzonego osobno na każdym z procesorów przy zrezygnowaniu z zastosowania enkoderów – czujników umieszczonych na kołach.

Wiele popularnych algorytmów mających za zadanie mapowanie i lokalizację w przestrzeni działa w oparciu o wskazania enkoderów, które pozwalają uzyskać informacje o kącie obrotu, co przy znanym promieniu pozwala z dużą dokładnością szacować przesunięcia liniowe i kątowe. Choć otrzymywane wyniki nie są idealne (odkształcenia, poślizgi kół, różnica w średnicy kół, po zastosowaniu odpowiednich algorytmów filtrujących pozwalają z dużą dokładnością ustalić domniemaną pozycję robota) [3].

W rozpatrywanym przypadku konieczna jest zmiana podejścia – zakłada się bowiem, że enkodery nie mogą zostać wykorzystane przy tworzeniu mapy. Sytuacja taka ma miejsce na przykład przy uruchomieniu algorytmu SLAM na dronie lub robocie kroczącym [4].

Do określenia przemieszczenia można użyć GPSu, jednak takie rozwiązanie nie sprawdzi się w przypadku zamkniętych pomieszczeń czy korytarzy.

Inną metodą jest użycie stereowizji jako podstawowego narzędzia pomiarowego [5], z możliwym, zaimplementowanym algorytmem optical flow [6].

W niniejszym artykule zaprezentowana zostanie metoda mapowania i lokalizacji niezależna od dodatkowych czujników, to jest operująca wyłącznie na danych uzyskanych z czujnika odległości. Oznacza to, że konieczne jest szybkie przetwarzanie dużej ilości danych.

## 2. Materiały i metody

Do badań wykorzystane zostały dwa komputery z procesorami Intel Core i5-8400, z kartą graficzną GeForce RTX 2080, mobilna platforma transportowa pozwalająca na swobodne przemieszczanie całego stanowiska badawczego po mierzonych korytarzach, stanowiących obszar testowy, czujnik odległości RPLidar oraz skaner 3d Kinect v2. Czujnik RPLidar jest to skaner laserowy działający w obszarze pełnego obrotu, o zasięgu do sześciu metrów, zapewniający od dziesięciu do piętnastu pomiarów na sekundę (rys. 1).

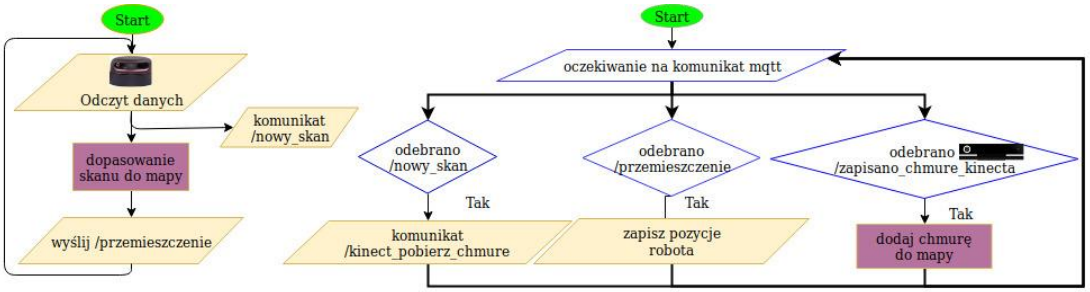
Rys. 1. RPLidar [7]

Jeden z komputerów przelicza dane otrzymywane z czujnika RPLidar, aktualizując swoją pozycję i dodając nowe punkty do mapy. Informacje o aktualnym położeniu robota przekazywane są na bieżąco do drugiego komputera, którego jedynym zadaniem jest pobieranie chmury punktów ze skanera 3D do późniejszej wizualizacji przebytej drogi. Komputery połączono za pomocą lokalnej sieci WiFi. Komunikują się za pomocą warstwy oprogramowania jaką stanowi lekki protokół transmisji danych MQTT (MQ Telemetry Transport) [8]. Schemat blokowy programów przedstawiono na rys. 2.

Program odpowiedzialny za mapowanie i lokalizacje został napisany w dwóch wersjach – pracującej w całości na CPU oraz takiej, w której kluczowe obliczenia zostały przeniesione na kartę graficzną.

## 3. Metoda

Zarówno program działający na CPU, jak i jego wersja wykorzystująca implementację GPU działają w bardzo podobny sposób. Po uruchomieniu, wykonywanych jest kilka skanów otoczenia, zakładając, że na początku robot nie będzie się poruszał. Skany te nakładane są na siebie i zapisywane w pamięci programu z zastrzeżeniem, że robot znajduje się na środku skanowanego obszaru. Rozdzielczość zapisywanej mapy wynosi 1 cm.




Rys. 2. Schemat blokowy programów

Aby dodawać kolejne fragmenty mapy potrzebna jest znajomość rotacji i przesunięcia robota względem posiadanej mapy. Program przeszukuje kwadrat o boku długości 20 cm, który następnie umiejscawia w ostatniej znanej pozycji robota. Uwzględniane są też możliwe rotacje w zakresie +/- 2 stopni, z dokładnością do jednego stopnia. Program ustala poziom dopasowania do konkretnego badanego punktu dzięki funkcji kosztu, która wyliczana jest w sposób następujący: dla każdej pary kąt–odległość uzyskanej podczas bieżącego skanowania wyliczana jest współrzędna XY, która dopasowywana jest do posiadanej mapy. Obliczone punkty, które nie mają swojej reprezentacji na utworzonej dotychczas mapie zwiększają koszt wprost proporcjonalnie do odległości do najbliższego istniejącego na mapie punktu, przy założeniu górnej granicy równej 10 cm.

Ostateczna wartość funkcji kosztu to suma wspomnianych odległości, obliczona dla każdej pary kąt–odległość otrzymanej z danego skanowania. Funkcja ta jest wyliczana dla każdego punktu w badanym kwadracie oddzielnie. Ponadto, aby uwzględnić możliwe rotacje, dla każdego z rozpatrywanych kątów program musi wyliczyć funkcje kosztu oddzielnie – przy zakresie +/- 2 stopnie rozpatrywany kwadrat musi zostać przebadany pięciokrotnie.

Różnica pomiędzy implementacją wyłącznie na CPU, a pracą z wykorzystaniem GPU widoczna jest w sposobie wyliczania funkcji kosztów. Jednostka, na której uruchamiany był program dysponuje 6-rdzeniowym procesorem. Zasadnym był więc podział części wyliczającej koszty na pięć równoległych wątków, z których każdy przeliczał kwadrat w otoczeniu ostatniej znanej pozycji robota dla innego kąta. Aby choć częściowo odciążyć procesor, każdy wątek zapamiętuje w którym punkcie rozpatrywanego w danej chwili kwadratu osiągnięto najniższą wartość funkcji kosztu oraz ile ona wynosi. Jeśli dla któregokolwiek z rozpatrywanych w danej iteracji programu punktów wartość funkcji kosztu będzie większa od zapamiętanej, program porzuca ten punkt i przechodzi do następnego. Takie podejście możliwe jest dzięki szeregowemu wyliczaniu kosztu kolejnych punktów.

Plusem takiego rozplanowania programu jest łatwość skalowania jeśli chodzi o zwiększanie maksymalnej dopuszczalnej rotacji. Żeby program mógł zwiększyć zakres badanego kąta o jeden stopień przy zachowaniu aktualnej szybkości trzeba przeznaczyć na to dwa dodatkowe rdzenie w procesorze. Czas wykonania pojedynczego przejścia programu zależny jest od ilości przetwarzanych punktów pomiarowych oraz od wielkości obszaru wokół robota, który ma zostać sprawdzony. Aby skrócić czas jednej iteracji programu można odrzucić część punktów pomiarowych, zawęzić sprawdzany obszar lub zastosować szybszy procesor.

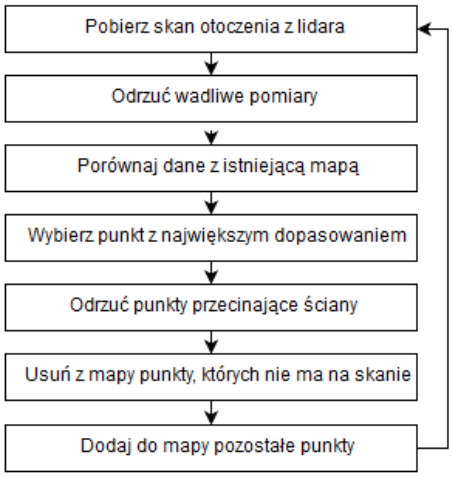
W implementacji przenoszącej część obliczeń na GPU funkcje kosztu wyliczane są w inny sposób. Do wydzielonego obszaru pamięci zapisywane są zestawy wcześniej przeliczonych współrzędnych XY dla każdego zestawu kąt-odległość uzyskanego podczas danego pomiaru,

przeliczone dla wszystkich kątów w badanym zakresie. Dla czujnika RPLidar, który dostarcza około 315 pomiarów na obrót przy założonym zakresie +/- 2 stopni, zapamiętywanych jest 1575 współrzędnych XY.

Następnie, współrzędne te przeliczane są równolegle na GPU – pojedynczy rdzeń CUDA sprawdza czy jeden zestaw współrzędnych XY wskazuje na punkt na zapamiętanej mapie, a jeśli nie – w jakiej odległości znajduje się najbliższy punkt. Następnie odpowiednie zestawy są sumowane, aby wyłonić przy jakim kącie funkcja kosztu jest najniższa. Zapamiętywana jest jej wartość i najlepszy kąt. Całość powtarzana jest dla każdego punktu na zadanym obszarze otaczającym ostatnią znaną pozycję robota. Stąd dla kwadratu o boku 20 cm obliczenia te wykonywane są 400 razy na jedną iterację programu.

Zaletą takiego rozplanowania programu jest prostota skalowania i dostosowywania programu do bieżących wymagań. Bardzo łatwo można tutaj manipulować zakresem przeliczanego kąta, jak również liczbą przeliczanych punktów pomiarowych. Dopóki liczba pomiarów na obrót, pomnożona przez liczbę sprawdzanych kątów nie przekroczy liczby posiadanych rdzeni CUDA (wykorzystywana karta graficzna dysponuje 2944 rdzeniami), nie będzie to miało znaczącego wpływu na czas wykonania pojedynczej iteracji programu. Żeby skrócić jej czas można zawęzić sprawdzany obszar lub zastosować kartę graficzną o większym taktowaniu.

Na koniec obie wersje programu znów posiadają cechy wspólne – raz na dziesięć iteracji program sprawdza, czy dotychczas wykryte punkty nadal mają swoje odzwierciedlenie w rzeczywistości. Dzięki temu możliwa jest drobna korekta fałszywie wykrytych ścian, pewnych przesunięć czy obiektów, które poruszały się wokół robota podczas mapowania przestrzeni. Schemat działania algorytmu tworzącego mapę przedstawiono na rys. 3.

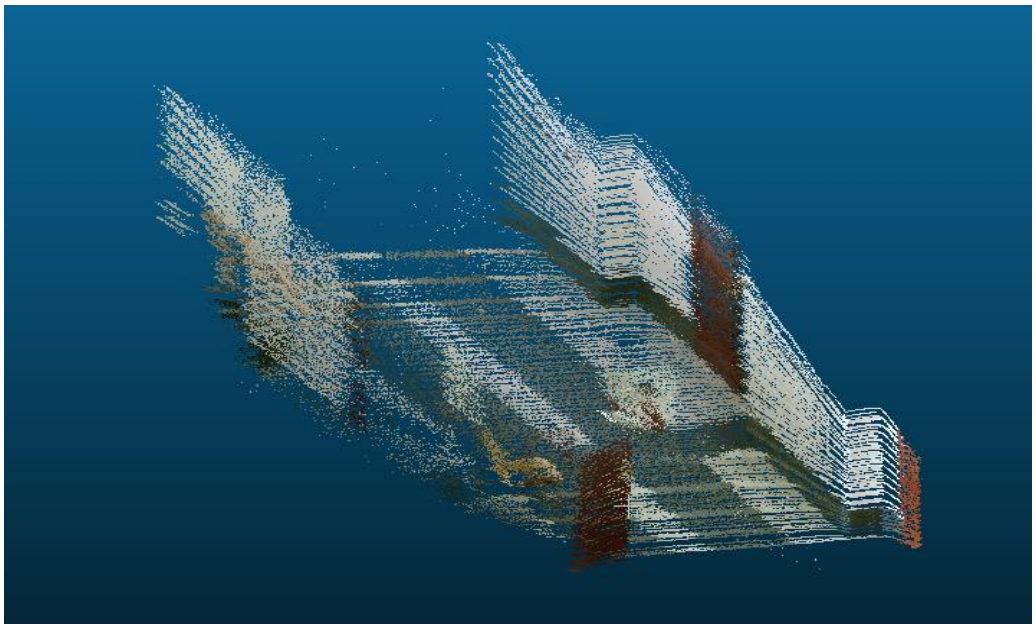



Rys. 3. Schemat blokowy algorytmu

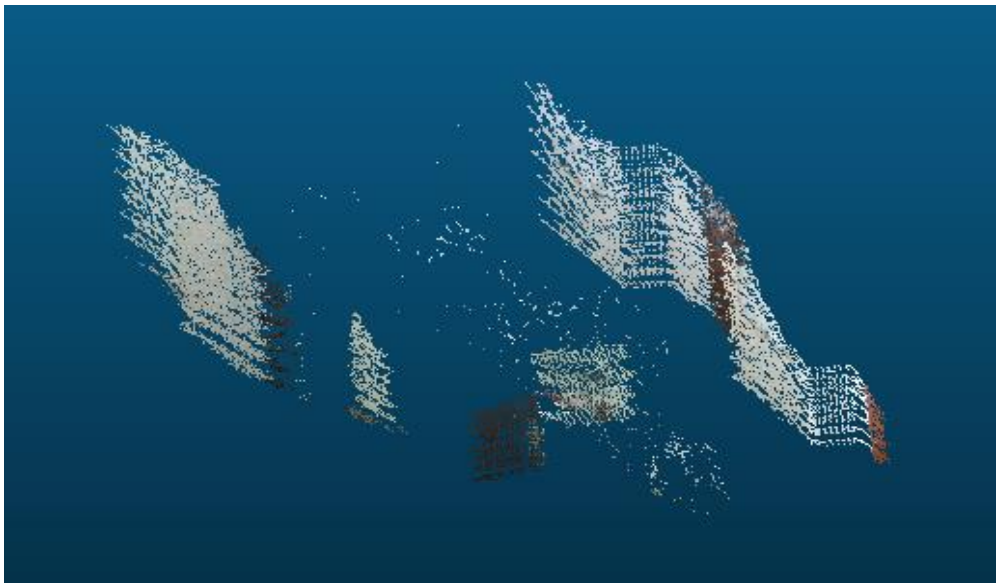
Podczas tworzenia mapy program na bieżąco raportuje swoją pozycję komputerowi obsługującemu skaner 3D. Chmury punktów rejestrowane przez czujnik Kinect są filtrowane wstępnie tak, aby zostało jedynie 16 linii obrazu z 424 dostępnych. Pozostawione linie posiadają kąt wpadającego światła w zakresie od -15 stopni do 15 stopni w odstępach co 2 stopnie. Filtrowanie wstępne ma zasymulować dane otrzymywane z czujnika Velodyne LiDAR PUCK VLP-16. Dodatkowo, chmura punktów zawiera informacje o kolorze, co znacząco zwiększa czytelność stworzonej z nich mapy.

Chmura, po filtrowaniu wstępnym poddawana jest filtrowaniu właściwemu – usuwane

są punkty, które znajdują się w bliskim sąsiedztwie punktów z poprzedniej chmury. Dane po takim przetworzeniu dodawane są do mapy. Skanowanie kończy się w momencie otrzymania komunikatu od pierwszego komputera.

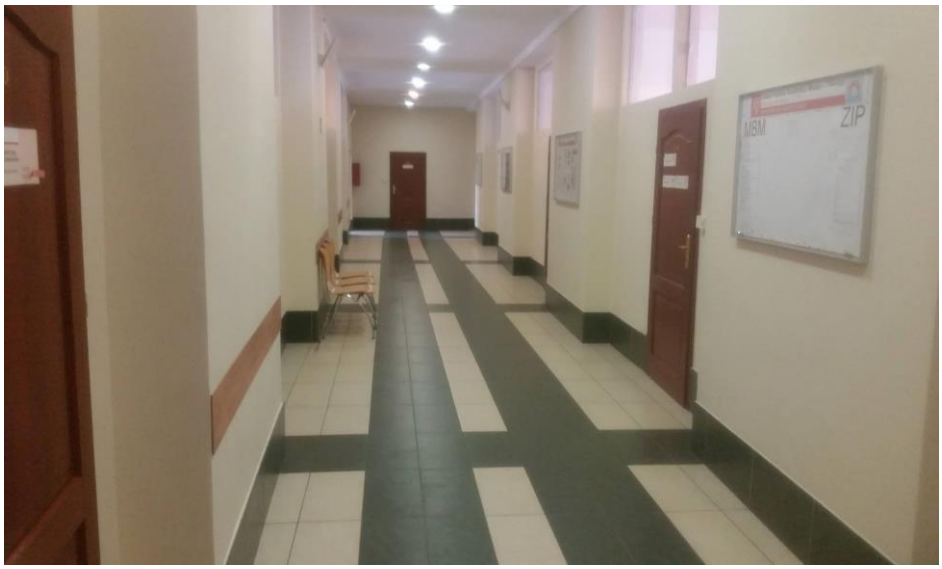


Rys. 4. Mapa przed filtracją



Rys. 5. Mapa po filtracji

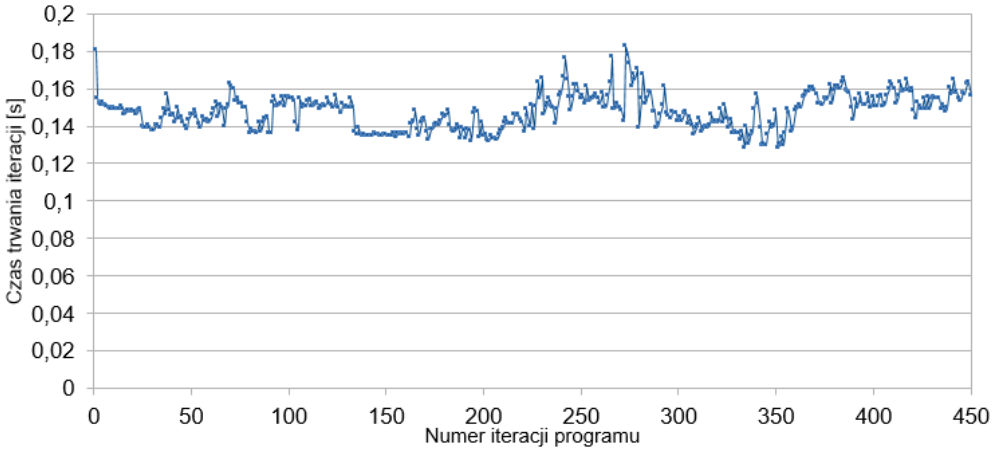
Ostatnim krokiem po zakończeniu skanowania jest post-processing mapy (rys. 4). Odpowiedzialny jest za to osobny program używający technologii CUDA w celu ponownego filtrowania punktów. Filtracja tutaj to voxelizacja punktów – w obszarze sześcianu o boku 4cm obliczana jest średnia współrzędna punktów. Następnie usuwane są wszystkie punkty w obszarze sześcianu a dodawany jest punkt o współrzędnych wcześniej obliczonej średniej. Proces filtrowania mapy złożonej z dwustu tysięcy punktów na czterdzieści tysięcy punktów to około 13 sekund. Mapę po filtracji obrazuje rys. 5. Dla porównania przedstawiono zdjęcie mapowanego obszaru – rys. 6.



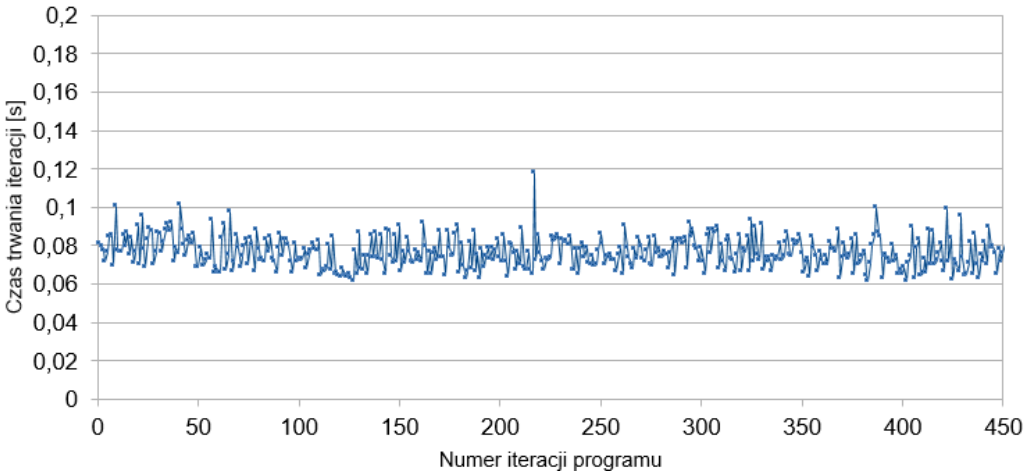

Rys. 6. Zdjęcie mapowanego obszaru

## 4. Wyniki

Poniżej zamieszczono wykresy przedstawiające czasy trwania kolejnych iteracji obu wersji programu oraz wygenerowane mapy. Aby wyeliminować wpływ czynników dodatkowych, takich jak czas potrzebny na wyświetlenie aktualnego stanu mapy, dodanie nowych punktów do mapy czy oczekiwanie na nowe dane pomiarowe, prezentowane wyniki dotyczą wyłącznie czasu działania części algorytmu odpowiedzialnej za przetworzenie surowych danych uzyskanych z czujnika do użytecznej postaci oraz zlokalizowanie robota w przestrzeni.




Rys. 7. Czas trwania iteracji przy wykorzystaniu wyłącznie CPU

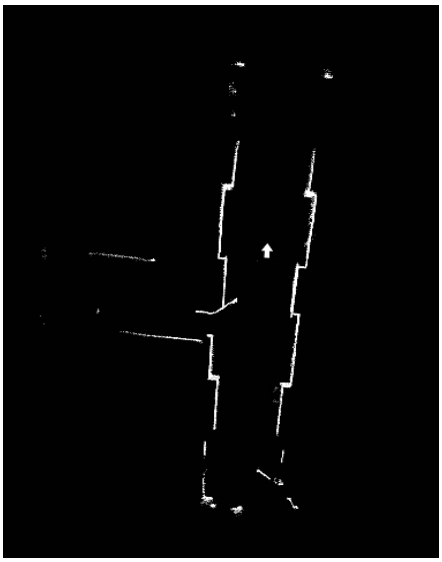



Rys. 8. Czas trwania iteracji przy wykorzystaniu GPU

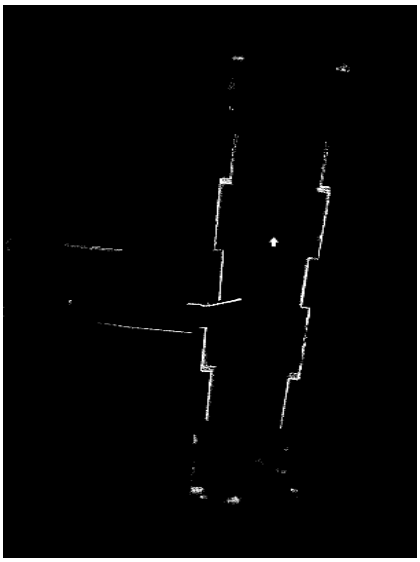
Wykres zamieszczony na rys. 7 przedstawia czas działania algorytmu wykorzystującego wyłącznie CPU. Czasy poszczególnych przeliczeń wahają się od około 0,13 do około 0,17 sekundy. Wykres przedstawiony na rys. 8 ukazuje czas trwania poszczególnych iteracji programu wykorzystującego GPU. Obliczenia trwają od około 0,06 do 0,1 sekundy.

Rys. 9 i 10 przedstawiają mapy uzyskane bezpośrednio z czujnika RPLidar. Podczas wykonywania map, oprócz ruchu liniowego stanowisko pomiarowe poddawane było rotacjom z zakresu około +- 5 stopni. Czas ruchu stanowiska pomiarowego nie był badany. Biała strzałka na mapie oznacza aktualną pozycję i orientację robota. Jak widać mapy są sobie równoważne i obie mogłyby zostać wykorzystane do nawigacji robota.

Podczas tworzenia mapy, drzwi jednego pokoju były otwarte, co zobrazowane zostało przez przerywany zarys widoczny w lewym fragmencie zbudowanej mapy. Krótki przejazd umożliwił wychwycenie nieprzysłoniętych ścian będących w zasięgu czujnika.



Rys. 9. Mapa otrzymana przy wykorzystaniu wyłącznie CPU

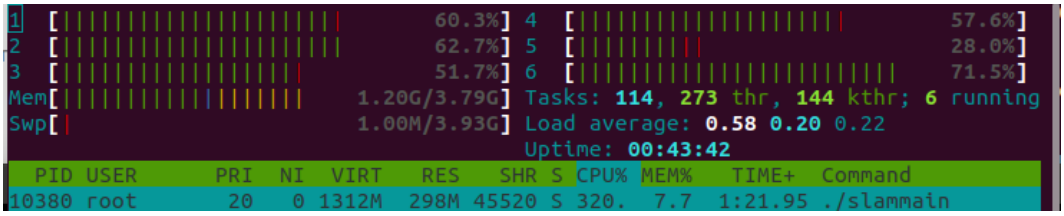


Rys. 10. Mapa otrzymana przy wykorzystaniu GPU

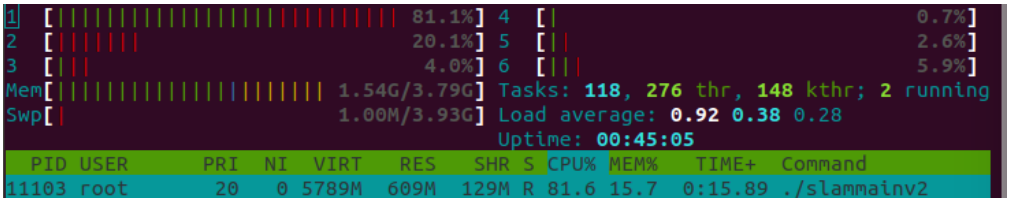
Rys. 11 oraz 12 obrazują zrzuty ekranu z programu *htop,* wyświetlającego informacje

o bieżącym zużyciu procesora. W przypadku programu działającego tylko na CPU widoczne jest ponad 50% zużycie pięciu z sześciu dostępnych rdzeni – ilustracja C. Program operujący na GPU obciąża w znaczący sposób tylko jeden rdzeń i część drugiego. Pozostałe 4 rdzenie są praktycznie nieużywane i mogą zostać wykorzystane do innych zadań.

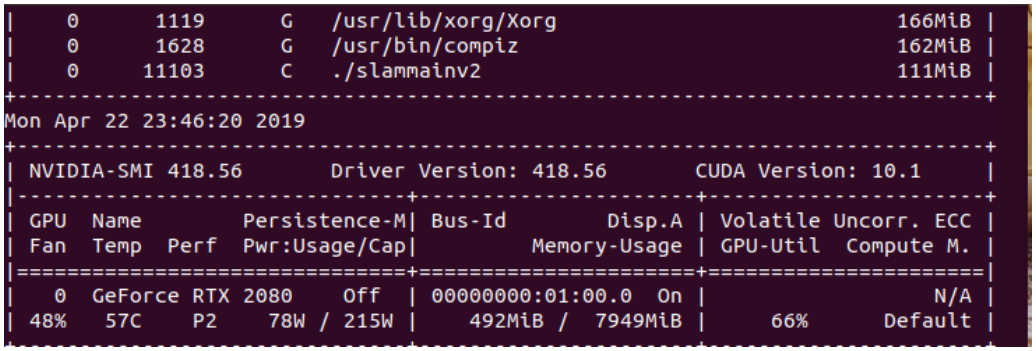
Rys. 13 przedstawia zrzut ekranu z programu *nvidia-smi,* ilustrujący obciążenie karty graficznej.




Rys. 11. Parametry zużycia procesora przy wykorzystaniu wyłącznie CPU




Rys. 12. Parametry zużycia procesora przy wykorzystaniu GPU




Rys. 13. Obciążenie karty graficznej

## 5. Wnioski

Zaproponowany algorytm mapujący pozwala osiągnąć zadowalające wyniki. Zarówno implementacja oparta na CPU, jak i ta wykorzystująca GPU generują poprawną mapę i dostarczają informacje do modułu tworzącego wizualizację 3D.

Aby przyspieszyć działanie tej wersji programu można wprowadzić komunikację pomiędzy poszczególnymi wątkami tak, by dzieliły się najlepszym uzyskanym wynikiem.

Jak widać na przedstawionych wykresach, algorytm wykorzystujący kartę graficzną do obliczeń działa niemal dwa razy szybciej. Czas wykonania poszczególnych przeliczeń jest znacząco niższy, co przełoży się bezpośrednio na większą maksymalną prędkość robota wykorzystującego przedstawione rozwiązania. Dodatkowo, algorytm wykorzystujący wyłącznie CPU zajmuje je niemal całkowicie, nie pozostawiając wolnych zasobów dla innych programów,

które mogłyby zostać uruchomione równocześnie na tej samej jednostce obliczeniowej. Wersja pracująca na GPU również obciąża procesor, jednak w znacznie niższym stopniu, zostawiając znaczną część czasu procesora niewykorzystaną, a co równie istotne – nie zajmuje nawet połowy z dostępnej mocy obliczeniowej karty graficznej.

Badania zrealizowane w ramach projektu nr DOB-2P/02/01/2018 finansowanego przez Narodowe Centrum Badań i Rowoju.

## Literatura

[1] Bailey T., Durrant-Whyte H.: "Simultaneous localization and mapping (SLAM): part II", IEEE Robotics & Automation Magazine, Sept. 2016, Vol. 13, Issue 3
[2] Żorski W., Makowski W.: „Use of CUDA technology in area of irregular patterns recognition" WAt Magazine, 2011, Vol. LX, nr 4, 201
[3] Mitka Ł., Ciszewski M., Kudriashov A., Buratowski T., Giergiel M.: „Robot with laser scanner for 2D mapping", Modelling in Engineering 2017 nr. 61.
[4] Rouveure R., Faure P., Monod M.O.: "Radar-based SLAM without odometric sensor", ROBOTICS 2010, International workshop of Mobile Robotics for environment/agriculture, Sep 2010, Clermont Ferrand, France. 9 p. hal-00583427
[5] Herath D.C., Kodagoda K.R.S., Dissanayake G.: "Stereo Vision Based SLAM Issues and Solutions", ARC Centre of Excellence for Autonomous Systems, University of Technology, Sydney, Australia
[6] Baronnier R.: "Optical Flow for SLAM", https://www.ensta-bretagne.fr/jaulin/rapport_pfe_romain_baronnier.pdf, 28.04.2019
[7] https://botland.com.pl/pl/skanery-laserowe/7036-skaner-laserowy-360-stopni-rplidar-a2m8.html, 28.04.2019
[8] Hillar G. C.: "MQTT Essentials - A Lightweight IoT Protocol", Pact Publishing Ltd., April 2017