Odwracanie macierzy szybką metodą iteracyjną w języku Julia

Kajetan Michalak, Krzysztof Pożoga, Filip Żarnowiec

28.01.25

1 Wstęp

Projekt polegał na implementacji iteracyjnego algorytmu odwracania macierzy, a następnie skorzystać z metod przyspieszania komputerowego wykonywania obliczeń i porównać uzyskane przyspieszenie. Obliczenia przyspieszamy poprzez dzielenie ich i rozdysponowywanie między dostępnymi zasobami takimi jak rdzenie procesora czy oddzielne maszyny liczące i wykonywanie ich równoległe.

W kontekście porównywania obliczeń równoległych algorytm ma dwie główne zalety. Pierwszą było łatwe zwiększanie złożoności obliczeniowej, która rośnie geometrycznie wraz ze wzrostem rozmiaru macierzy, generowanej losowo. Drugą było łatwe dzielenie obciążenia obliczeniowego – macierze można dzielić na mniejsze lub oddzielnie mnożyć pary wierszy i kolumn, później łącząc wyniki w całość.

W języku Julia dostępne są różne środowiska i biblioteki umożliwiające równoległe przetwarzanie danych, takie jak:

- **MPI (Message Passing Interface)**: Standard komunikacji międzyprocesowej, który umożliwia równoległe przetwarzanie na wielu węzłach obliczeniowych. MPI jest szczególnie przydatne w środowiskach klastrów obliczeniowych.
- **OpenCL (Open Computing Language)**: Framework do programowania równoległego, który umożliwia wykonywanie obliczeń na różnych urządzeniach, takich jak procesory, karty graficzne (GPU) oraz akceleratory.
- **CUDA (Compute Unified Device Architecture)**: Platforma programistyczna firmy NVIDIA, która umożliwia wykorzystanie mocy obliczeniowej kart graficznych do równoległego przetwarzania danych.
- **Distributed**: Wbudowane środowisko w Julii, które umożliwia łatwe tworzenie i zarządzanie równoległymi obliczeniami na wielu procesach, zarówno na jednym, jak i wielu węzłach obliczeniowych.

2 Iteracyjny algorytm odwracania macierzy

Iteracyjne metody odwracania macierzy opierają się na idei stopniowego poprawiania przybliżenia macierzy odwrotnej. Jednym z podejść jest wykorzystanie macierzy rezydualnej, która jest definiowana jako różnica między macierzą jednostkową a iloczynem macierzy wejściowej i jej aktualnego przybliżenia odwrotności. Formalnie, dla macierzy $A \in \mathbb{R}^{n \times n}$, macierz rezydualna R_k w k-tej iteracji jest dana wzorem:

$$R_k = I - AX_k$$

gdzie X_k jest aktualnym przybliżeniem macierzy odwrotnej A^{-1}. Celem algorytmu jest minimalizacja normy macierzy rezydualnej, co prowadzi do poprawy przybliżenia X_k. W kolejnych iteracjach, przybliżenie X_k jest aktualizowane zgodnie z wzorem:

$$X_{k+1} = X_k + \alpha R_k$$

gdzie α jest współczynnikiem skalarnym, który może być dobierany w celu optymalizacji zbieżności algorytmu.

3 Zrównoleglenie obliczeń

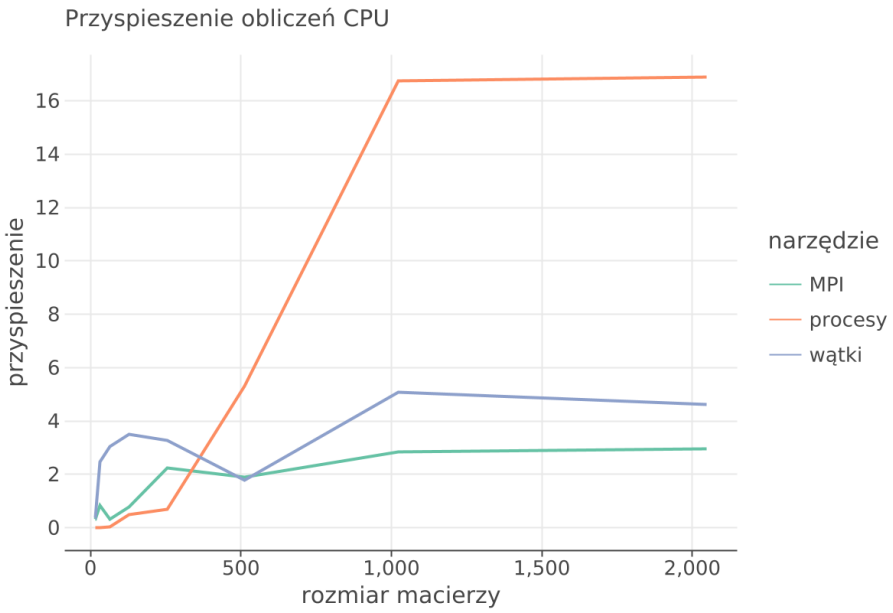
3.1 CPU

Dla operacji mnożenia macierzy, której złożoność wynosi $O(N^3)$, koszt pozostałych operacji może zostać pominięty dla odpowiednio dużych macierzy. Teoretycznie, program powinien przyspieszyć wprost proporcjonalnie do liczby wykorzystanych rdzeni. W naszym przypadku oczekiwany był 16-krotny wzrost wydajności względem wersji sekwencyjnej. Jednak ze względu na koszt komunikacji między wątkami oraz ograniczenia prędkości pamięci, wzrost ten nie jest proporcjonalny, a zwiększenie liczby wątków może nawet wydłużyć czas wykonania.

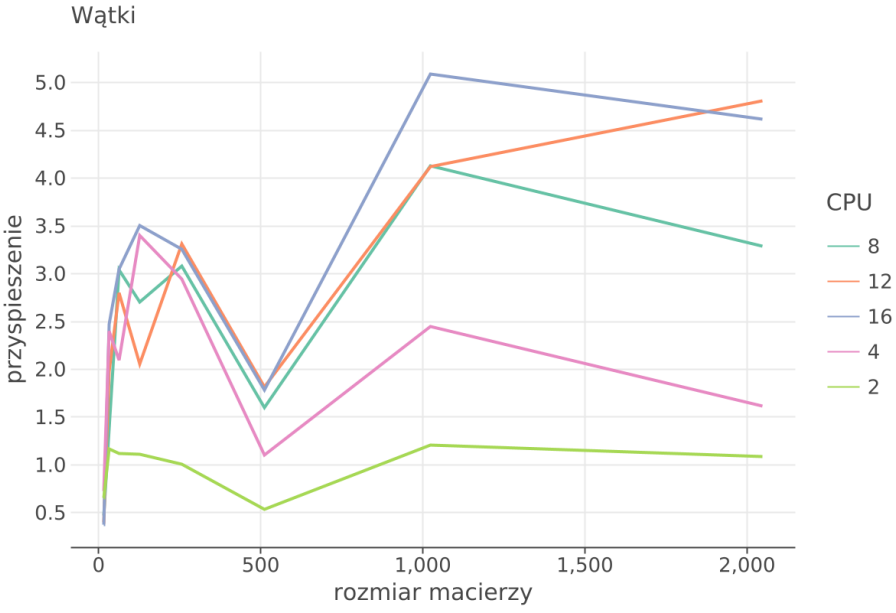
3.1.1 Wątki

Pierwszym sposobem przyspieszenia obliczeń było rozdzielenie ich i rozdysponowanie na wątki, czyli oddzielne zadania które system operacyjny rozdziela między fizyczne jednostki obliczeniowe (rdzenie) procesora. Obliczenia w poszczególnych wątkach wykonywane są niezależnie od siebie nawzajem, ale współdzielą ze sobą jedną przestrzeń adresową, czyli pracują na tym samym obszarze pamięci. Implementacja wątków polegała na podziale macierzy na wiersze po równo pomiędzy dostępnych wątków i wykonanie mnożenia tylko dla tych kilku wierszy przez każdy proces.

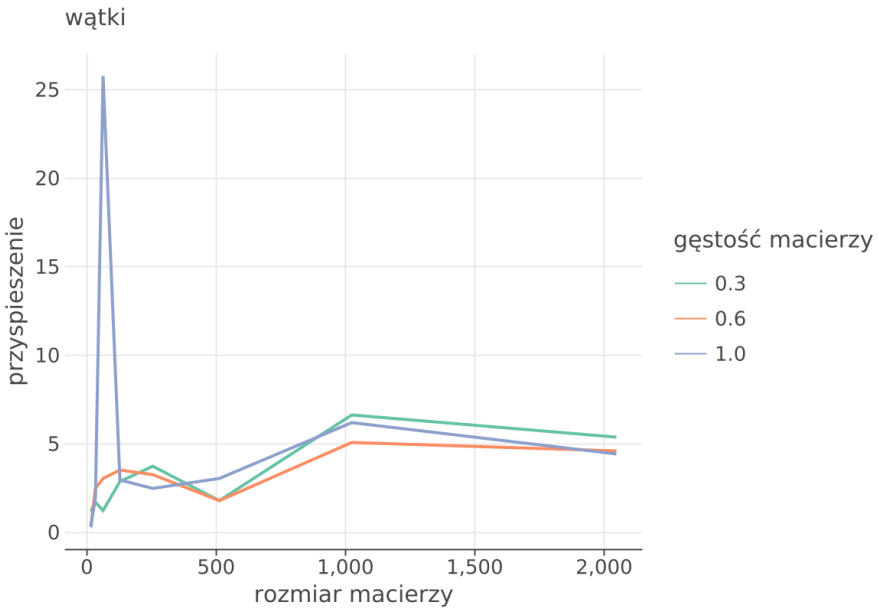
Do deklaracji i obsługi wątków posłużyła składnia “Threads.@threads” znajduje się w pakiecie Distributed.



Rysunek 1: Przyspieszenie dla obliczeń z wykorzystniem CPU.



Rysunek 2: Uzyskane przyspieszenie dla wątków.

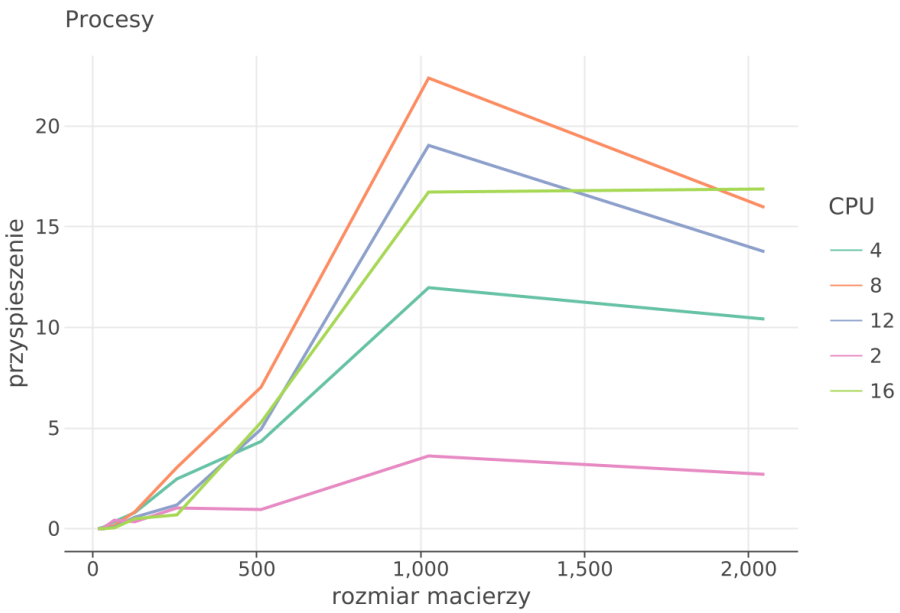


Rysunek 3: Uzyskane przyspieszenie dla wątków.

```julia
function parallel_row_multiply(A, B)
    n, m = size(A)
    _, p = size(B)

    C = Matrix{Float64}(undef, n, p)

    Threads.@threads for i in 1:n
        row_A = view(A, i, :)
        for j in 1:p
            col_B = view(B, :, j)
            C[i, j] = sum(row_A[k] * B[k, j] for k in 1:m)
        end
    end

    return C
end
```

Listing 1: Fragment pliku threads.jl.

3.1.2 Procesy rozproszone



Rysunek 4: Uzyskane przyspieszenie dla procesów.

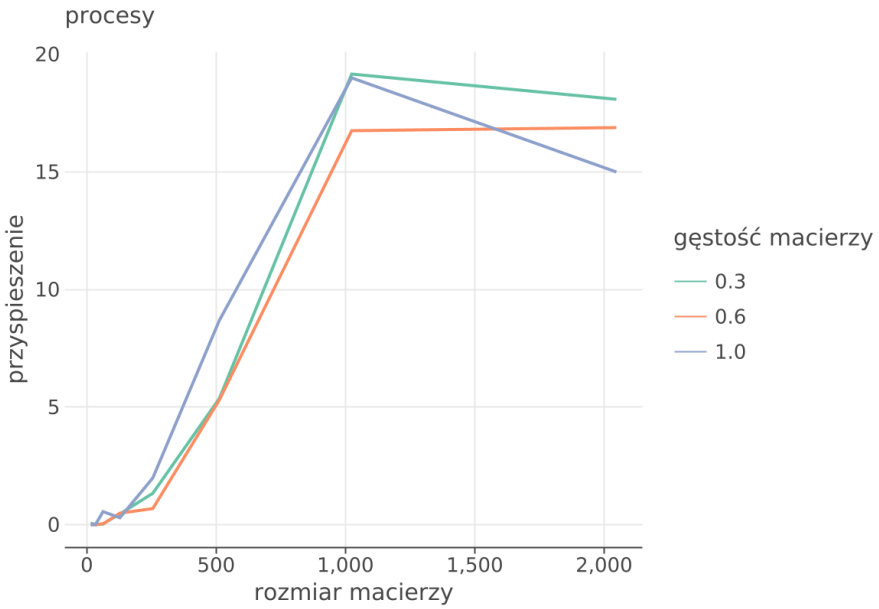
Drugim sposobem przyspieszenia obliczeń był podział na procesy rozproszone. W przeciwieństwie do wątków, będących stosunkowo lekkimi zadaniami bez własnej pamięci, przystosowanymi do działania na jednej maszynie, procesy rozproszone to bardziej pełne, odizolowane od siebie nawzajem niezależne zadania o własnych przestrzeniach adresowych. Takie rozwiązanie zakłada wydzielenie zasobów na każdy proces, co zwiększa koszt ich tworzenia, ale umożliwia prowadzenie równoległych obliczeń poza obrębem jednego urządzenia, np. na klastrach czy w chmurach, choć środowisko Julii nie rozróżnia, czy procesy mają działać na jednej maszynie, czy rozproszone między wieloma.

```
function parallel_row_multiply(A, B)
    n, m = size(A)
    _, p = size(B)

    C = SharedArray{Float64}(n, p)

    @sync @distributed for i in 1:n
        row_A = A[i, :]
        for j in 1:p
            C[i, j] = sum(row_A[k] * B[k, j] for k in 1:m)
```

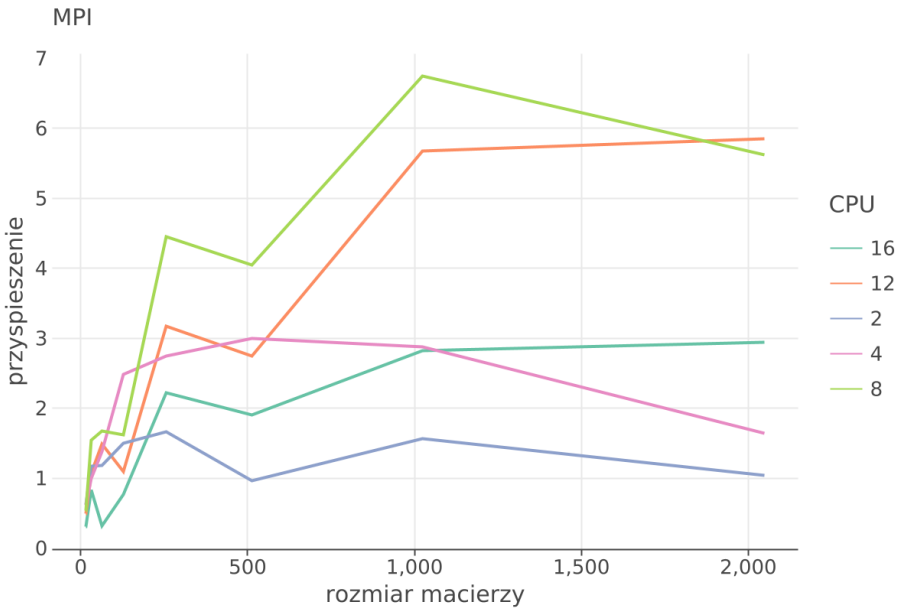
```
        end
    end

    return C
end
```

Listing 2: Fragment pliku distributed.jl.

Implementacja procesów rozproszonych była bardzo podobna do implementacji wątków, macierz znowu rozdzielano na wiersze przydzielane do danych procesów. Różnica polegała na wykorzystaniu struktury SharedArrays zamiast zwykłej tablicy, podczas rekonstruowania wynikowej macierzy. SharedArrays tworzy tablicę dostępną dla wszystkich podprocesów i dba o dostęp do zasobów w przypadku wystąpienia konfliktów. Dalej do deklaracji i obsługi wątków posłużyła składnia "@sync @distributed", podobnie jak obsługa wątków pobrana z pakietu Distributed.



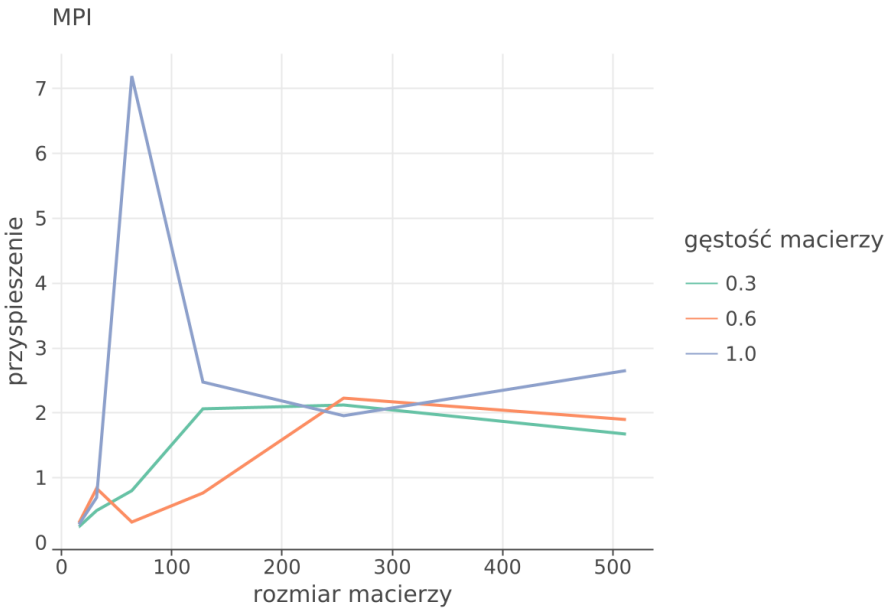
Rysunek 5: Uzyskane przyspieszenie dla procesów.



Rysunek 6: Uzyskane przyspieszenie dla MPI.

3.1.3 MPI

MPI w przeciwieństwie do pozostałych podejść nie umożliwia wykonania tylko części programu na osobnym wątku / procesie. Korzystając z MPI uruchamiamy wiele procesów, z których każdy wykonuje cały kod programu. Każdy proces ma swój zestaw zmiennych i różnią się tylko wartością zmiennej "rank" będącą numerem danego procesu. Bazując na tej zmiennej możemy podzielić macierz i obliczenia podobnie jak w poprzednich rozwiązaniach, a następnie zsynchronizować wyniki korzystając z funkcji Gather oraz Bcast. Ze względu na duży koszt inicjalizacji MPI dla małych macierzy czas osiągnięty z użyciem MPI nie różni się zbytnio od czasu wykonania metody sekwencyjnej. Poprawa jest jednak widoczna, jeśli porównamy ten czas z czasem wykonania MPI dla jednego procesu.



Rysunek 7: Uzyskane przyspieszenie dla MPI.

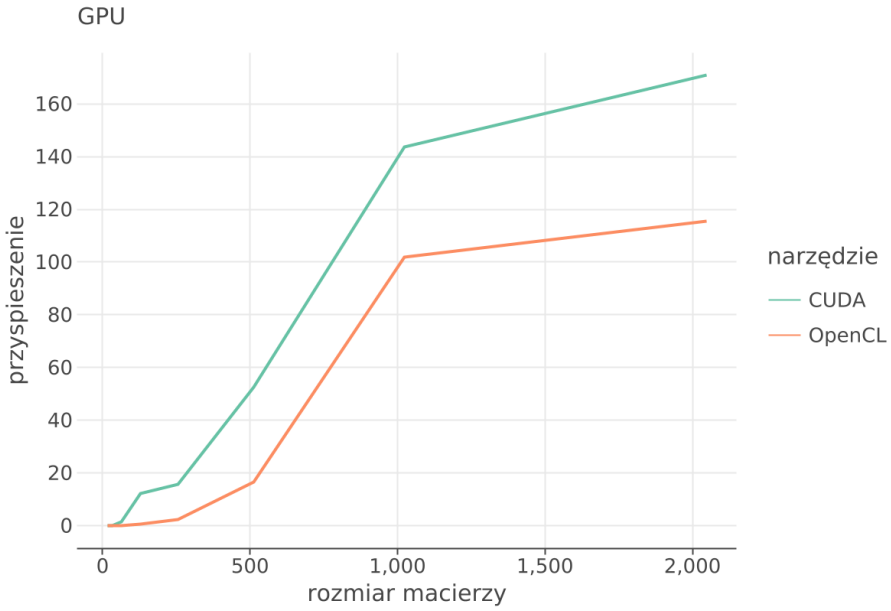
3.2 GPU

Obliczenia z wykorzystaniem jednostek graficznych (GPU) stanowią kluczową technologię przyspieszania operacji macierzowych dzięki masywnej równoległości oferowanej przez architekturę kart graficznych. W przeciwieństwie do CPU, które posiadają kilkadziesiąt rdzeni, nowoczesne GPU dysponują tysiącami prostych jednostek obliczeniowych, optymalizowanych pod kątem jednoczesnego wykonywania wielu operacji na dużych zbiorach danych. W kontekście odwracania macierzy, operacje takie jak mnożenie macierzy czy aktualizacja macierzy rezydualnej mogą być efektywnie zrównoleglone na GPU.

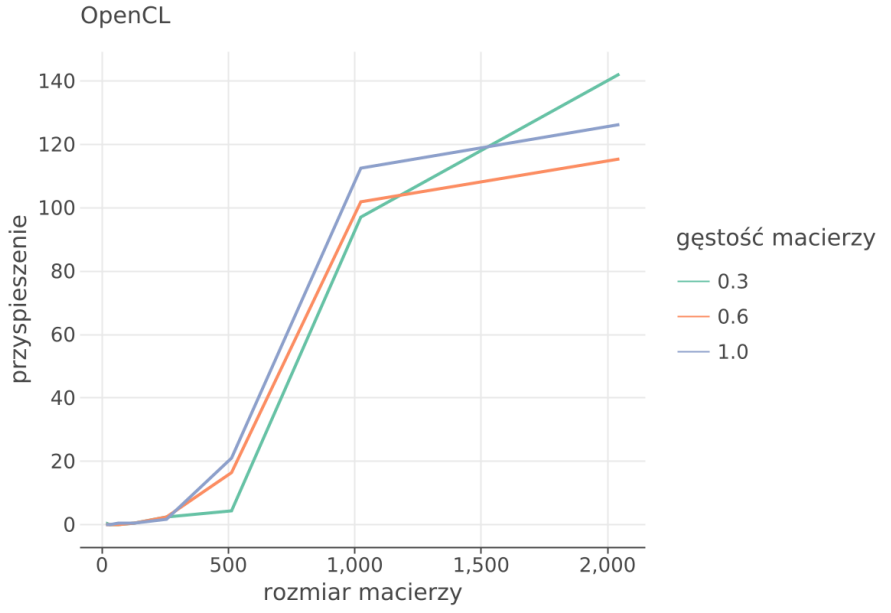
3.2.1 OpenCL

OpenCL (Open Computing Language) to otwarty standard umożliwiający programowanie heterogenicznych systemów zawierających procesory CPU, GPU oraz inne akceleratory.

```
kernel_source = """
__kernel void mmul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float* A,
```



Rysunek 8: Przyspieszenia uzyskana dla GPU.



Rysunek 9: Uzyskane przyspieszenie dla OpenCL.

```
    __global float* B,
    __global float* C)
{
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp;
    if ((i < Ndim) && (j < Mdim))
    {
        tmp = 0.0f;
        for (k = 0; k < Pdim; k++)
            tmp += A[i*Ndim+k] * B[k*Pdim+j];
        C[i*Ndim+j] = tmp;
    }
}
"""
prg  = cl.Program(source=kernel_source) |> cl.build!
mmul = cl.Kernel(prg, "mmul")

function mat_mul_openCl(A::Matrix{Float32}, B::Matrix{Float32}):: Matrix{Float32
    m, n = size(A)
    _, p = size(B)

    d_a = CLArray{Float32}(undef, m * n)
    d_b = CLArray{Float32}(undef, n * p)
    d_c = CLArray{Float32}(undef, m * p)
    copyto!(d_a, A)
    copyto!(d_b, B)
    C = zeros(Float32, p * m)

    global_size = (m, p)
    local_size = (min(m, 16), min(p, 16))

    cl.queue!(:profile) do
        evt = clcall(mmul, Tuple{Int32, Int32, Int32, Ptr{Float32}, Ptr{Float32}
                    m, n, p, d_a, d_b, d_c; global_size, local_size)
        wait(evt)
        cl.copy!(C, d_c)
    end

    return reshape(C, p, m)
end
```

Listing 3: Fragment pliku opecl.jl.

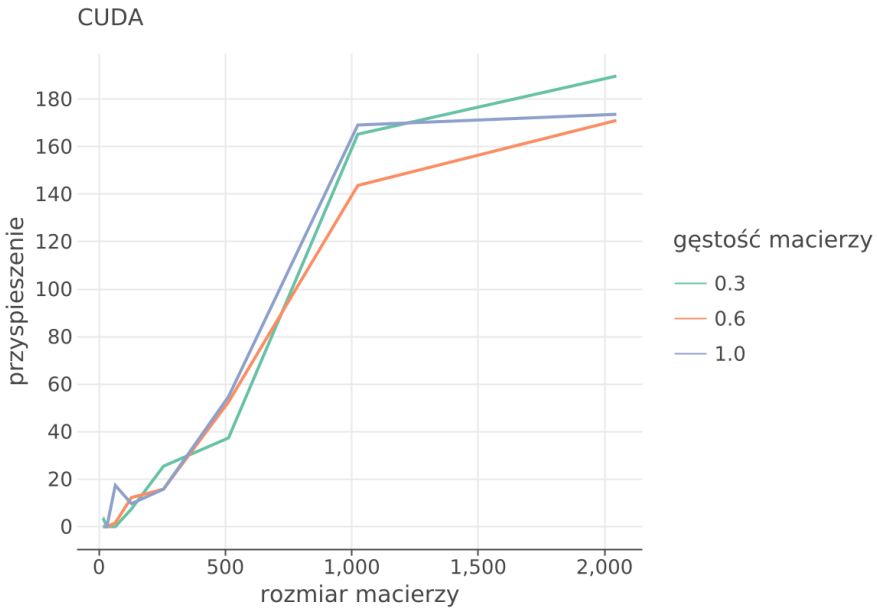
3.2.2 CUDA

CUDA to platforma firmy NVIDIA dedykowana programowaniu kart graficznych tej marki. W Julii pakiet `CUDA.jl` abstrahuje niskopoziomowe detale, umożliwiając operacje na macierzach GPU przy użyciu składni zbliżonej do standardowych tablic. W przeciwieństwie do OpenCL, CUDA zapewnia głębszą integrację z bibliotekami optymalizacyjnymi (np. cuBLAS), co często przekłada się na wyższą wydajność.

W implementacji macierz A jest konwertowana do postaci CUDA array przy użyciu funkcji `CuArray`. Operacje takie jak mnożenie macierzy są delegowane do zoptymalizowanych kernelów CUDA, co minimalizuje narzut komunikacyjny.

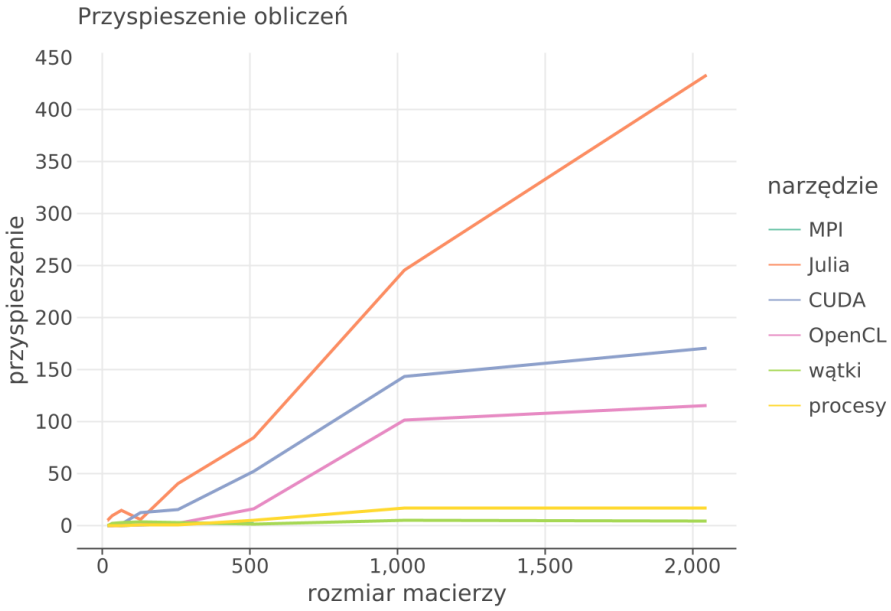
```
function mat_mul_cuda(A::Matrix{Float32}, B::Matrix{Float32})::Matrix{Float32}
    m, n = size(A)
    _, p = size(B)

    d_a = CuArray{Float32}(undef, m, n)
    copyto!(d_a, A)
    d_b = CuArray{Float32}(undef, n, p)
    copyto!(d_b, B)
    d_c = CuArray{Float32}(undef, m, p)

    threads_per_block = (16, 16)  # 16x16 threads per block
    blocks_per_grid = (ceil(Int, m / threads_per_block[1]), ceil(Int, p / thread

    @cuda threads=threads_per_block blocks=blocks_per_grid matmul_kernel(m, n, p

    C = Array(d_c)
    return C
end

function matmul_kernel(Mdim, Ndim, Pdim, A, B, C)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    j = threadIdx().y + (blockIdx().y - 1) * blockDim().y

    if i <= Mdim && j <= Pdim
        tmp = 0.0f0
        for k in 1:Ndim
            tmp += A[i + (k - 1) * Mdim] * B[k + (j - 1) * Ndim]
        end
        C[i + (j - 1) * Mdim] = tmp
    end

    return
end
```

Listing 4: Fragment pliku cuda.jl.



Rysunek 10: Uzyskane przyspieszenie dla CUDA.

4 Podsumowanie



Rysunek 11: Zestawienie wszystkich osiągniętych przyspieszeń.

Iteracyjne algorytmy odwracania macierzy, takie jak te wykorzystujące macierz rezydualną, są potężnym narzędziem w obliczeniach numerycznych, szczególnie w przypadku dużych macierzy. W połączeniu z metodami równoległymi dostępnymi w Julii, takimi jak MPI, OpenCL, CUDA oraz Distributed, możliwe jest znaczące przyspieszenie obliczeń, co czyni te metody atrakcyjnymi dla zastosowań wymagających wysokiej wydajności. Na wykresie 11 zostało narysowane przyspieszenie jakie uzyskalibyśmy używając wbudowanej funkcji Julii ‘inv‘, nie mniej jednak nie jest to ten sam algorytm.

Wykorzystanie GPU pozwoliło osiągnąć istotne przyspieszenie algorytmu, szczególnie dla dużych macierzy. CUDA okazała się bardziej efektywna niż OpenCL w testowanym środowisku. Konsekwentnie przewyższa OpenCL o około 10-15% dzięki lepszej optymalizacji sterowników i wsparciu sprzętowemu. Należy jednak zauważyć, że dla małych macierzy (poniżej 256x256) narzut związany z transferem danych pomiędzy CPU a GPU dominuje nad czasem obliczeń, co prowadzi do spadku wydajności.

5 Załącznik

Tabela 1: Wyniki czasowe dla różnych narzędzi

Rozmiar	Gęstość	Czas [s]	Narzędzie
16	0.3	2.357 773 232	MPI
16	0.6	0.000 480 914	MPI
16	1.0	0.000 452 207	MPI
32	0.3	0.001 700 234	MPI
32	0.6	0.001 130 74	MPI
32	1.0	0.001 212 406	MPI
64	0.3	0.007 466 136	MPI
64	0.6	0.019 840 435	MPI
64	1.0	0.008 784 743	MPI
128	0.3	0.031 761 219	MPI
128	0.6	0.090 567 704	MPI
128	1.0	0.026 329 32	MPI
256	0.3	0.291 963 187	MPI
256	0.6	0.231 874 348	MPI
256	1.0	0.245 763 76	MPI
512	0.3	3.365 180 107	MPI
512	0.6	3.048 924 586	MPI
512	1.0	3.095 659 559	MPI
16	0.3	0.712 204 263	Julia
16	0.6	$2.9263 \cdot 10^{-5}$	Julia
16	1.0	$2.2582 \cdot 10^{-5}$	Julia
32	0.3	$9.0326 \cdot 10^{-5}$	Julia
32	0.6	$9.5908 \cdot 10^{-5}$	Julia
32	1.0	$9.1296 \cdot 10^{-5}$	Julia
64	0.3	0.000 458 743	Julia
64	0.6	0.000 438 757	Julia
64	1.0	0.000 627 414	Julia
128	0.3	0.004 204 031	Julia
128	0.6	0.011 106 725	Julia
128	1.0	0.001 994 829	Julia
256	0.3	0.017 817 226	Julia
256	0.6	0.012 844 398	Julia
256	1.0	0.014 259 057	Julia
512	0.3	0.156 404 377	Julia
512	0.6	0.068 724 192	Julia
512	1.0	0.075 225 804	Julia
1024	0.3	0.687 923 982	Julia

Kontynuacja na następnej stronie

Tabela 1 – Kontynuacja

Rozmiar	Gęstość	Czas [s]	Narzędzie
1024	0.6	0.591 214 471	Julia
1024	1.0	0.484 816 861	Julia
2048	0.3	3.237 261 68	Julia
2048	0.6	3.081 839 316	Julia
2048	1.0	3.345 443 433	Julia
16	0.3	0.581 310 646	sekwencyjnie
16	0.6	0.000 148 505	sekwencyjnie
16	1.0	0.000 126 317	sekwencyjnie
32	0.3	0.000 834 153	sekwencyjnie
32	0.6	0.000 940 05	sekwencyjnie
32	1.0	0.000 839 022	sekwencyjnie
64	0.3	0.005 987 907	sekwencyjnie
64	0.6	0.006 347 909	sekwencyjnie
64	1.0	0.063 205 781	sekwencyjnie
128	0.3	0.065 682 766	sekwencyjnie
128	0.6	0.069 654 186	sekwencyjnie
128	1.0	0.065 109 441	sekwencyjnie
256	0.3	0.619 682 588	sekwencyjnie
256	0.6	0.516 469 818	sekwencyjnie
256	1.0	0.479 772 475	sekwencyjnie
512	0.3	5.619 057 941	sekwencyjnie
512	0.6	5.797 433 605	sekwencyjnie
512	1.0	8.229 096 77	sekwencyjnie
1024	0.3	153.231 875 677	sekwencyjnie
1024	0.6	145.096 377 773	sekwencyjnie
1024	1.0	160.282 845 676	sekwencyjnie
2048	0.3	1550.143 646 686	sekwencyjnie
2048	0.6	1335.386 705 195	sekwencyjnie
2048	1.0	1422.547 190 814	sekwencyjnie
16	0.3	0.170 270 4	CUDA
16	0.6	0.005 698 6	CUDA
16	1.0	0.004 126 5	CUDA
32	0.3	0.074 266 2	CUDA
32	0.6	0.008 781 7	CUDA
32	1.0	0.004 669 5	CUDA
64	0.3	0.205 787	CUDA
64	0.6	0.004 204 3	CUDA
64	1.0	0.003 642 8	CUDA
128	0.3	0.008 767 7	CUDA
128	0.6	0.005 659 6	CUDA
128	1.0	0.006 616 1	CUDA
256	0.3	0.024 234 3	CUDA

Kontynuacja na następnej stronie

Tabela 1 – Kontynuacja

Rozmiar	Gęstość	Czas [s]	Narzędzie
256	0.6	0.032 786 4	CUDA
256	1.0	0.030 239 1	CUDA
512	0.3	0.150 910 2	CUDA
512	0.6	0.110 307 7	CUDA
512	1.0	0.150 506 4	CUDA
1024	0.3	0.927 846 3	CUDA
1024	0.6	1.010 025 3	CUDA
1024	1.0	0.948 066 9	CUDA
2048	0.3	8.170 140 1	CUDA
2048	0.6	7.811 118	CUDA
2048	1.0	8.194 249 6	CUDA
16	0.3	1.055 348 1	OpenCL
16	0.6	0.079 282 3	OpenCL
16	1.0	0.093 758 4	OpenCL
32	0.3	0.136 208 3	OpenCL
32	0.6	0.095 096 5	OpenCL
32	1.0	0.078 579 3	OpenCL
64	0.3	0.120 173 5	OpenCL
64	0.6	0.125 672 1	OpenCL
64	1.0	0.118 789 2	OpenCL
128	0.3	0.151 661 7	OpenCL
128	0.6	0.154 465 4	OpenCL
128	1.0	0.158 793 6	OpenCL
256	0.3	0.256 191 9	OpenCL
256	0.6	0.221 521 1	OpenCL
256	1.0	0.264 947 4	OpenCL
512	0.3	1.277 544	OpenCL
512	0.6	0.350 926 9	OpenCL
512	1.0	0.392 200 5	OpenCL
1024	0.3	1.577 474	OpenCL
1024	0.6	1.424 054 7	OpenCL
1024	1.0	1.423 308 6	OpenCL
2048	0.3	10.898 781 8	OpenCL
2048	0.6	11.564 530 5	OpenCL
2048	1.0	11.257 500 6	OpenCL
16	0.3	0.486 198 603	wątki
16	0.6	0.000 395 828	wątki
16	1.0	0.000 390 828	wątki
32	0.3	0.000 492 127	wątki
32	0.6	0.000 38	wątki
32	1.0	0.000 431 074	wątki
64	0.3	0.004 940 784	wątki

Kontynuacja na następnej stronie

Tabela 1 – Kontynuacja

Rozmiar	Gęstość	Czas [s]	Narzędzie
64	0.6	0.002 083 967	wątki
64	1.0	0.002 452 348	wątki
128	0.3	0.022 971 497	wątki
128	0.6	0.019 894 479	wątki
128	1.0	0.022 087 794	wątki
256	0.3	0.165 128 423	wątki
256	0.6	0.158 547 046	wątki
256	1.0	0.194 693 463	wątki
512	0.3	3.164 272 012	wątki
512	0.6	3.247 940 436	wątki
512	1.0	2.717 175 805	wątki
1024	0.3	23.099 593 465	wątki
1024	0.6	28.508 693 832	wątki
1024	1.0	25.813 517 938	wątki
2048	0.3	287.087 960 38	wątki
2048	0.6	289.465 987 54	wątki
2048	1.0	319.780 516 516	wątki
16	0.3	9.015 051 069	procesy
16	0.6	0.067 128 571	procesy
16	1.0	0.127 409 614	procesy
32	0.3	0.134 201 325	procesy
32	0.6	0.140 946 119	procesy
32	1.0	0.174 218 379	procesy
64	0.3	0.132 656 499	procesy
64	0.6	0.202 198 699	procesy
64	1.0	0.113 975 05	procesy
128	0.3	0.142 215 133	procesy
128	0.6	0.140 115 766	procesy
128	1.0	0.229 387 488	procesy
256	0.3	0.458 288 211	procesy
256	0.6	0.750 487 998	procesy
256	1.0	0.241 956 446	procesy
512	0.3	1.046 022 811	procesy
512	0.6	1.095 305 15	procesy
512	1.0	0.946 096 929	procesy
1024	0.3	7.996 448 589	procesy
1024	0.6	8.667 795 961	procesy
1024	1.0	8.435 315 526	procesy
2048	0.3	85.744 941 651	procesy
2048	0.6	79.067 052 116	procesy
2048	1.0	94.782 029 899	procesy