

Równoległe implementacje metod numerycznych – laboratorium

Preskrypt

na prawach rękopisu

Marek Nałęcz (ćw. 1–3)
Jacek Gambrych (ćw. 1, 3)
Gustaw Mazurek (ćw. 4–5)

Zakład Teorii Obwodów i Sygnałów
Instytut Systemów Elektronicznych
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Warszawa, 8 stycznia 2025

Spis treści

1	Zapoznanie się ze środowiskiem programistycznym CUDA	5
1.1	Mój pierwszy (na laboratorium) program CUDA	5
1.1.1	Utworzenie projektu	5
1.1.2	Upiększenie programu	6
1.1.3	Pierwsze testy	7
1.1.4	Kolejne testy	8
1.1.5	Pomiar czasu wykonania brutto	8
1.1.6	Pomiar czasu wykonania netto	9
1.1.7	Wydruki kontrolne	9
1.1.8	Nadchodzi zmienny przecinek	10
1.1.9	Profilowanie programu	10
1.1.10	Odpluskwanie programu	11
1.1.11	Ścieżka kompilacji	12
1.1.12	Podglądanie rozkazów maszynowych	13
1.1.13	Kompilacja z linii komendy	13
1.1.14	Refaktoryzacja kodu	14
1.2	Program rysujący śliczne fraktale	14
1.2.1	Wersja CPU	14
1.2.2	Wersja GPU	18
2	Optymalizacja programów w języku CUDA C	21
2.1	Filtracja zakłóceń sygnału akustycznego	21
2.1.1	Współczynnik CGMA dla filtracji FIR	22
2.1.2	Implementacja filtracji FIR na CPU	22
2.1.3	Wybór sygnału akustycznego	27
2.1.4	Niszczanie i odtwarzanie sygnału akustycznego	27
2.1.5	Uruchomienie filtracji FIR na CPU dla sygnału akustycznego	29
2.1.6	Pierwsza (naiwna) implementacja w środowisku CUDA	29
2.1.7	Lekko zmieniona implementacja w środowisku CUDA	31
2.1.8	Umieszczenie współczynników filtru w pamięci stałej	32
2.1.9	Przetwarzanie sygnału techniką „kafelkową”	33
2.1.10	Statyczny rząd filtru i rozwinięcie pętli	35
2.1.11	Dodatkowe próby optymalizacji	36

3	Biblioteki wysokiego poziomu	38
3.1	Całkowanie metodą Monte Carlo	38
3.1.1	Sformułowanie problemu	38
3.1.2	Prototypowy program na procesor CPU – obliczanie objętości kuli	40
3.1.3	Przeniesienie programu na procesor GPU	46
3.1.4	Generacja liczb pseudolosowych za pomocą biblioteki CURAND	46
3.1.5	Zastąpienie liczb pseudolosowych quasi-losowymi	48
3.1.6	Liczenie objętości innych brył	49
3.2	Odtwarzanie kompresyjnie spróbkowanego sygnału	50
3.2.1	Sformułowanie problemu	50
3.2.2	Implementacja w języku MATLAB	52
3.2.3	Implementacja w języku CUDA C z wykorzystaniem biblioteki CU-BLAS	54
4	Implementacja operacji numerycznych w układzie FPGA	60
4.1	Płyta ewaluacyjna Basys 3	60
4.2	Komunikacja płyty z komputerem PC	62
4.2.1	Komunikacja przez USB – interfejs JTAG	62
4.2.2	Blok komunikacji: rimlab01	62
4.2.3	Przykładowy plik ograniczeń (.XDC)	64
4.2.4	Komunikacja w środowisku Matlab: skrypt fpga.m	65
4.2.5	Środowisko projektowe Vivado	66
4.3	Pierwszy projekt w języku VHDL dla układu FPGA	66
4.3.1	Tworzenie projektu i dodanie źródeł	66
4.3.2	Główny plik projektu – fpgatop.vhd	67
4.3.3	Kompilacja projektu	69
4.3.4	Konfiguracja układu FPGA na płycie ewaluacyjnej	70
4.3.5	Uruchomienie algorytmu przetwarzania	72
4.4	Badanie algorytmu filtracji FIR	72
4.4.1	Implementacja filtru FIR w języku VHDL	73
4.4.2	Implementacja mnożenia w komórkach logicznych	77
4.4.3	Implementacja filtru o zmiennych współczynnikach	78
4.4.4	Filtr o zm. współczynnikach, implementacja w blokach DSP	79
5	Biblioteki obliczeniowe dla układów FPGA	81
5.1	Biblioteka IP Catalog	81
5.2	Synteza filtru FIR (moduł FIR Compiler)	82
5.2.1	Implementacja filtru FIR w strukturze bezpośredniej	84
5.2.2	Implementacja filtru FIR w strukturze transponowanej	84
5.2.3	Wyprowadzenia modułu FIR	85
5.3	Implementacja alg. FFT (moduł <i>Fast Fourier Transform</i>)	86
5.3.1	Architektura strumieniowa FFT	87
5.3.2	Architektura FFT z dekompozycją o podstawie 4	87
5.3.3	Architektura FFT z dekompozycją o podstawie 2	87
5.3.4	Uproszczona architektura FFT z dekompozycją o podstawie 2	88
5.3.5	Wyprowadzenia bloku FFT	88

5.4	Szybka aproksymacja modułu liczby zespolonej	90
5.4.1	Implementacja algorytmu w kodzie VHDL	90
5.4.2	Implementacja rejestru opóźniającego	91
5.5	Badanie bloku filtracji FIR	92
5.5.1	Utworzenie projektu	92
5.5.2	Utworzenie bloku filtracji FIR	92
5.5.3	Instalowanie komponentu FIR w projekcie	100
5.5.4	Kompilowanie projektu	102
5.5.5	Badanie zrównoleglonej architektury MAC	102
5.5.6	Badanie sekwencyjnej architektury MAC	103
5.6	Badanie implementacji transformaty Fouriera	104
5.6.1	Utworzenie bloku FFT	104
5.6.2	Instalowanie bloku FFT w projekcie	107
5.6.3	Kompilowanie projektu	108
5.6.4	Badanie algorytmu FFT o podstawie 4	108
5.6.5	Badanie algorytmu FFT o podstawie 2	110
5.6.6	Implementacja obliczeń modułu liczby zespolonej	111

Rozdział 1

Zapoznanie się ze środowiskiem programistycznym CUDA

Podczas laboratorium nie trzeba pisać sprawozdania, ale trzeba „chwalić się” prowadzącemu każdym wykonanym zadaniem i możliwie szybko zgłaszać ew. trudności w jego wykonaniu. Przed laboratorium należy wykonać w domu zadanie 1.2.1. Uwaga: w laboratorium jest dostępny Internet, więc dane i ew. kod tego zadania można przynieść z domu albo na nośniku USB, albo zapisać na swoim koncie na jakimś, najlepiej wydziałowym, serwerze.

W celu przyspieszenia wykonywania laboratorium zaleca się kopiowanie fragmentów kodu bezpośrednio z niniejszego PDF-a do schowka Windows, a następnie wklejanie ich do edytora tekstowego lub do okienka komend. Dłuższe fragmenty kodu dołączono do tego dokumentu jako tekstowe załączniki, wyróżnione rysunkiem pineski na marginesie. Niektóre polecenia linii komendy z powodu swojej długości zostały w tekście dokumentu złamane na wiele wierszy i przez to nie wklejałyby się poprawnie do okienka komend. Dlatego zostały one w całości powtórzone w formie komentarzy PDF-a, oznaczonych żółtymi notesikami na marginesie. Można je stamtąd skopiować do schowka i wkleić w okienku komend. Uwaga: w domyślnej na laboratorium przeglądarce PDF-ów (PDFXCview) trzeba włączyć oglądanie komentarzy klawiszem *Ctrl-8*.

1.1 Mój pierwszy (na laboratorium) program CUDA

1.1.1 Utworzenie projektu

Dokonyamy kompilacji pierwszego programu CUDA spod zintegrowanego środowiska programistycznego Microsoft Visual Studio 2022:

1. Menu: *Start — Wszystkie programy — Visual Studio 2022*.
2. Menu: *Get Started — Create a new project — CUDA 12.6 Runtime* lub *File — New — Project — CUDA 12.6 Runtime*.
3. Wybieramy nazwę projektu i jego lokalizację (w katalogu roboczym). Zgadza się na domyślną nazwę rozwiązania (solution) i nie tworzymy dla niego odrębnego katalogu. Klikamy *OK*. Otwiera się projekt z jednym szablonowym plikiem `kernel.cu`.

4. Ustawiamy konfigurację (*Debug/Release*), a następnie platformę *x64* (należy pamiętać, aby ten i wszystkie kolejne projekty tworzyć w trybie 64-bitowym!).
5. Wybierając z menu *Build — Build Solution*¹ budujemy projekt, po czym przez naciśnięcie *Ctrl-F5* uruchamiamy go (jest to zalecana metoda uruchamiania programu, zwłaszcza w przypadku dokonywania pomiarów czasu jego działania). Aby obejrzeć wyniki podczas pracy debuggera (uruchamianego przez naciśnięcie *F5*) albo ustawiamy pułapkę na wyjściu z *main*-a, albo (typowa sztuczka!) tuż przed wyjściem z *main*-a dodajemy linię:

```
if (IsDebuggerPresent()) getchar();
```

a na początku programu dołączamy plik nagłówkowy:

```
#include <windows.h>
```

1.1.2 Upiększenie programu

Nasz program zadziałał i dał prawidłowe wyniki. Na początek spróbujmy go zrozumieć i przeanalizować, co on robi.

1. Program powinien być w pierwszej kolejności czytelny. Dlatego upraszczamy obsługę błędów, dołączając na początku pomocniczy plik nagłówkowy:

```
"helper_cuda.h"
```

i zamieniając wszystkie sekcje (wywołujące funkcje CUDA Runtime) o postaci:

```
cudaStatus = cudaXxxx(...);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaXxxx failed!");
    goto Error;
}
```

na postać:

```
checkCudaErrors(cudaXxxx(...));
```

Usuwamy etykietę *Error*: i deklarację zmiennej *cudaStatus*; typ funkcji *addWithCuda* zmieniamy na *void* i odpowiednio modyfikujemy jej prototyp i wywołanie przez funkcję *main*.

2. Aby upewnić się, na jakiej karcie graficznej pracujemy, warto zaraz po wywołaniu funkcji *cudaSetDevice* zbadać właściwości (np. nazwę i możliwości obliczeniowe) wybranego urządzenia, umieszczając w kodzie następujący fragment programu:

¹**Uwaga!** Z uwagi na zdrażające się czasami problemy z integracją najnowszych wersji Visual Studio z CUDA zmiany w plikach źródłowych .cu nie zawsze są „widziane” przez środowisko Visual Studio i konieczne trzeba się upewnić, czy uruchamiamy najnowszą wersję programu. W wypadku gdy zmiany w plikach .cu nie wymuszają ponownej kompilacji, zaleca się dalej zamiast *Build — Build Solution* używać pozycji menu *Build — Rebuild Solution*.

```
int devID;
checkCudaErrors(cudaGetDevice(&devID));
cudaDeviceProp props;
checkCudaErrors(cudaGetDeviceProperties(&props, devID));
printf("Device %d: \"%s\", compute capability = %d.%d\n",
       devID, props.name, props.major, props.minor);
```

3. W polu *Solution Explorer* klikamy prawym guzikiem myszy na nazwę projektu i wybieramy *Properties* (ostatnią pozycję). W nowo otwartym oknie jako opcję *Configuration* wybieramy pozycję *All configurations*. Do opcji *Configuration Properties — VC++ Directories — Include Directories* dodajemy pozycję `$(NVCUDASAMPLES_ROOT)/common/inc`. Jest ona niezbędna do znalezienia pliku `helper_cuda.h`.
4. Ponownie budując i uruchamiając program sprawdzamy, czy nadal działa on poprawnie.
5. Analizujemy strukturę programu, identyfikując przede wszystkim: alokację pamięci dla GPU, kopiowanie danych do karty graficznej, uruchomienie jądra na GPU, kopiowanie wyników obliczeń z powrotem do pamięci CPU i zwolnienie pamięci.
6. Ponieważ wkrótce będziemy przerabiali program tak, aby działał on na liczbach zmiennoprzecinkowych, przygotujmy już teraz jego kod do łatwej zmiany podstawowego typu danych. Jak można to uczynić (**#define**, **typedef**, **template**)?
7. Raz jeszcze budujemy i uruchamiamy program aby upewnić się, że nic nie popsuliśmy.

1.1.3 Pierwsze testy

Program jest na pewno dobry, a nawet najlepszy na świecie (przecież napisał go sam Komputer!), ale na wszelki wypadek trochę go potestujemy.

1. Na początek zamiast inicjować tablice `a` i `b` przy ich deklarowaniu, wypełniamy je pętlami **for** ze zmienną pętlową `i`:

```
a[i] = i;
b[i] = i * 10000;
```

Wymaga to oczywiście usunięcia modyfikatorów **const** z deklaracji tablic `a` i `b`. Zamiast wydruku wszystkich pięciu wyników dodawania i ich sprawdzania ręcznie, sprawdzamy ich poprawność kolejną pętlą **for**, weryfikującą, czy na pewno zachodzi warunek

```
c[i] == i * 10001.
```

Ponownie budujemy i uruchamiamy program. Powinien on działać prawidłowo.

2. Zwiększamy rozmiar wektorów `arraySize` do 2000 i ponawiamy eksperyment. Tym razem powinniśmy otrzymać komunikat o błędzie *InvalidConfiguration* w pierwszej instrukcji po wywołaniu funkcji jądra. (Nie zwraca ona żadnego statusu błędu (jest typu **void**), więc jej status bada się właśnie oddzielnym wywołaniem funkcji `cudaGetLastError()`.) Okazuje się, że przekroczyliśmy maksymalny rozmiar bloku wątków.
3. Tym razem zatem pracę rozdzielamy pomiędzy wiele bloków wątków, np.:

```
<<<(size + 255) / 256, 256>>>
```

Odpowiedniej zmianie musi też ulec wyliczanie indeksu wektora wewnątrz funkcji jądra!

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Ponownie budujemy i uruchamiamy program. Powinien on już działać prawidłowo.

1.1.4 Kolejne testy

Jeśli upewniliśmy się, że program działa poprawnie, to pozbadźmy się złudzeń!

1. W oknie konsoli tekstowej wchodzimy do podkatalogu `x64/Debug` w katalogu roboczym i uruchamiamy nasz program, podając jego nazwę jako argument programu `compute-sanitizer --tool memcheck`. Powinniśmy, niestety, dostać komunikat o naruszeniu ochrony pamięci.
2. Dokładamy do funkcji jądra dodatkowy argument `size` określający rozmiar tablicy danych oraz zabezpieczenie:

```
if (i < size)
```

po czym ponawiamy sprawdzenie dostępu do pamięci. Tym razem powinno już być wszystko dobrze.

1.1.5 Pomiar czasu wykonania brutto

Teraz już głęboko wierzymy, że nasz program nie ma katastrofalnych błędów. Ma tylko co najwyżej pewne niedokładności. A zatem możemy spróbować pomierzyć czas wykonania programu. Na początek zmierzmy czas wykonania obliczeń brutto, tzn. z alokacją pamięci, transferami danych itp.

1. Na początku programu wstawiamy:

```
#include "helper_timer.h"
```

2. Wewnątrz funkcji `main` wstawiamy w odpowiednie miejsca:

```
StopWatchInterface *timer = NULL;
...
sdkCreateTimer(&timer);
sdkStartTimer(&timer);
    addWithCuda(...);
sdkStopTimer(&timer);
...
printf("Processing time: %f ms\n", sdkGetTimerValue(&timer));
sdkDeleteTimer(&timer);
```

3. Budujemy i uruchamiamy program, notując (zaskakująco duży) czas jego wykonania.

Aby uzyskać bardziej wiarygodne wyniki pomiaru czasu wykonania brutto, warto program uruchomić z linii komendy kilkakrotnie (3...10 razy) i wziąć pod uwagę tylko *najlepszy* wynik (*najkrótszy* czas).

1.1.6 Pomiar czasu wykonania netto

W kolejnym kroku zmierzmy czas wykonania obliczeń netto, tzn, tylko wywołania jądra CUDA, za pomocą mechanizmu zdarzeń CUDA.

1. Wewnątrz funkcji `addWithCuda` wstawiamy w odpowiednie miejsca:

```
cudaEvent_t start, stop;
float elapsedTime;
...
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));
checkCudaErrors(cudaEventRecord(start, 0));
    addKernel<<<...>>>(...);
    checkCudaErrors(cudaGetLastError());
checkCudaErrors(cudaEventRecord(stop, 0));
checkCudaErrors(cudaEventSynchronize(stop));
checkCudaErrors(cudaEventElapsedTime(&elapsedTime, start, stop)
    );
checkCudaErrors(cudaEventDestroy(start));
checkCudaErrors(cudaEventDestroy(stop));
...
printf("Kernel-only time: %f ms\n", elapsedTime);
```

2. Budujemy i uruchamiamy program, odnotowując (zaskakująco dużą) różnicę czasu jego wykonania brutto a netto!

Aby uzyskać bardziej wiarygodne wyniki pomiaru czasu wykonania netto, warto (niezależnie od opisanego w poprzednim punkcie kilkukrotnego uruchomienia całego programu z linii komendy) kilkakrotnie w pętli uruchomić jądro (3...10 razy) i analogicznie uwzględnić tylko *najlepszy* wynik (*najkrótszy* czas).

1.1.7 Wydruki kontrolne

Bardzo prymitywną, ale niezwykle skuteczną techniką uruchamiania programów jest użycie wydruków kontrolnych. Mamy taką możliwość również w przypadku funkcji wykonywanych na GPU, ale należy wtedy pamiętać o bardzo wielu równocześnie wykonywanych wątkach – potencjalnie z każdego z nich możemy dostać jakiś wydruk. Można też wybrać identyfikator bloku i/lub wątku, do którego mają się ograniczyć wydruki.

1. Wewnątrz funkcji jądra umieszczamy wydruk:

```
printf("Block(%u,%u,%u), thread(%3u,%u,%u), "
      "clk=%u: %d<-%d+%d\n",
      blockIdx.x, blockIdx.y, blockIdx.z,
      threadIdx.x, threadIdx.y, threadIdx.z,
      clock(), c[i], a[i], b[i]);
```

(Oczywiście powyższy wydruk można zubożyć lub wzbogacić w zależności od aktualnych potrzeb).

2. Zmniejszamy tymczasowo rozmiar wektorów do 60 i uruchamiamy program. Jak mają się odczyty zegara GPU funkcją `clock()` do podziału bloków na sploty (warp) po 32 wątki? Jaki jest czas wykonania pojedynczej funkcji jądra?
3. Zwiększamy z powrotem rozmiar wektorów do 2000. Jakie teraz są czasy wykonania (różnice zegara `clock()`) poszczególnych splotów wątków? Aby uzyskać jaśniejszy obraz sytuacji, można dokonywać wydruków tylko z wybranych wątków.

1.1.8 Nadchodzi zmienny przecinek

Po pomiarze czasu wykonania programu aż chciałoby się obliczyć, z jaką wydajnością (GFLOP/s) on pracował. Odpowiedź jest prosta: z zerową! Dlaczego? Bo nie wykonywał w ogóle operacji zmiennoprzecinkowych...

1. Zmieniamy typ tablic na float i wypełniamy je:

```
a[i] = (float) i;
b[i] = (float) i * 0.0001f;
```

odpowiednio modyfikując również pętlę sprawdzania poprawności wyników. Tym razem zaczynamy działać na liczbach zmiennoprzecinkowych, więc możemy już mówić o FLOPS-ach. Użycie zmiennego przecinka jednak wiąże się z możliwością wystąpienia błędów skończonej dokładności reprezentacji liczb. Jako miarę błędu przyjmujemy maksymalne co do modułu odchylenie od wartości wyliczonej przez CPU, odpowiednio modyfikując funkcję sprawdzania i wyświetlania poprawności wyników. Na chwilę zakomentujemy wydruki kontrolne w funkcji jądra, aby upewnić się o poprawności (małym błędzie) wyników. Potem ponownie je odkomentowujemy i badamy czasy wykonania poszczególnych splotów wątków.

1.1.9 Profilowanie programu

Pomiary czasu wykonania programu i jego poszczególnych fragmentów można także (i to w sposób znacznie czytelniejszy i łatwiejszy do analizy) prześledzić za pomocą profilera. Zanim to uczynimy, zbudujemy wersję produkcyjną (*Release*) naszej aplikacji.

1. Zmieniamy konfigurację projektu na *Release*, zablokowujemy wydruki kontrolne ze wszystkich wątków i budujemy projekt, po czym uruchamiamy program, notujemy oba czasy jego wykonania (brutto i netto) oraz porównujemy z czasami dla konfiguracji uruchomieniowej (*Debug*).
2. Dla najnowszych kart graficznych i najnowszych wersji CUDA będziemy wykorzystywali program NVIDIA Nsight System, natomiast dla starszych kart graficznych i starszych wersji CUDA będzie to program Visual Profiler.
3. NVIDIA Nsight System

- (a) Uruchamiany program z menu: *Start — Wszystkie programy — NVIDIA Corporation — Nsight System*.

- (b) W nowo otwartym okienku wybieramy *File — New Project*. W oknie projektu wybieramy *Select target for profiling... — Localhost connection Desktop-....* Oprogramowanie *Nsight System* posiada bardzo duże możliwości, z których większość wykracza poza tematykę laboratorium, dlatego w kolejnym oknie odznaczamy wszystkie domyślnie zaznaczone opcje i zaznaczamy jedynie *Collect CUDA trace*, w zakładce *Target application* wskazujemy na naszą aplikację, po czym naciskamy przycisk *Start*.
- (c) Obserwujemy (w razie potrzeby w powiększeniu) wykres czasowy (timeline) i porównujemy widoczne tam czasy z czasem wydrukowanym przez nasz program.
- (d) Przez chwilę eksplorujemy wszystkie możliwości profilera, a zwłaszcza wyniki dokonanej przez niego analizy przyczyn słabej (jego zdaniem!) wydajności naszego programu.

4. NVIDIA Visual Profiler

- (a) Uruchamiamy program z menu: *Start — Wszystkie programy — NVIDIA Corporation — Visual Profiler*.
- (b) W nowo otwartym okienku profilera wybieramy *File — New Session*, po czym w okienku, które się pojawi, wybieramy nazwę pliku (*File*), wskazując naszą aplikację z katalogu *x64/Release*. W kolejnym okienku *Executable Properties* zaznaczamy wszystkie możliwe opcje oprócz *Enable unified memory profiling* i naciskamy *Finish*. Obserwujemy (w razie potrzeby w powiększeniu) wykres czasowy (timeline) i porównujemy widoczne tam czasy z czasem wydrukowanym przez nasz program.
- (c) Przez chwilę eksplorujemy wszystkie możliwości profilera, a zwłaszcza wyniki dokonanej przez niego analizy przyczyn słabej (jego zdaniem!) wydajności naszego programu.

1.1.10 Odpluskwanie programu

Na ogół lepszą metodą poszukiwania błędów w programie niż zastosowanie wydruków kontrolnych jest użycie debuggera (*Nsight*).

1. Uaktywniamy konfigurację *Debug*. Ustawiamy pułapkę na linii:

```
c[i] = a[i] + b[i];
```

a jeśli chcemy oglądać wartości elementów tablicy *c* – także na linii następnej. Potem uruchamiamy debugger. W tym celu z menu zintegrowanego środowiska wybieramy *Extensions — Nsight — Start CUDA Debugging (Next-Gen)*. Po zatrzymaniu programu na pułapce sprawdzamy pracę krokową debuggera. Obserwujemy wyróżnienie kolorem czerwonym zmiennych, których wartości zmieniły się od ostatniego wyświetlenia. Sprawdzamy przełączanie się pomiędzy splotami wątków (*Extensions — Nsight — Previous/Next Active Warp*).

2. Obserwujemy działanie specjalizowanych okienek debuggera: *Extensions — Nsight — Windows — Warp Info* oraz *Extensions — Nsight — Windows — Warp Watch*. Następnie przeglądamy zawartość systemu pomocy dotyczącą zastawiania pułapek w kodzie

GPU (*Extensions — Nsight — Help — Local Help — NVIDIA Nsight Visual Studio Edition — CUDA Debugger — Control GPU Execution — Set GPU Breakpoints*). Sprawdzamy np. działanie pułapek warunkowych, w tym wykorzystujących makra debuggera `@threadIdx` i `@blockIdx`. Można je zastawiać, klikając prawym guzikiem myszy na założoną pułapkę, a następnie wybierając *Condition* albo *Hit Count*.

3. Po otwarciu dodatkowego okienka z zawartością pamięci *Debug — Windows — Memory — Memory 1* możemy np. uchwycić myszą którąś z tablic `a`, `b` czy `c` i przeciągnąć ją do tego okienka, aby zobaczyć jej zawartość. Uwaga: nie możemy niestety tej zawartości zmieniać ani z okienka *Locals*, ani z okienka *Memory*.

1.1.11 Ścieżka kompilacji

Spróbujemy teraz zrozumieć ścieżkę kompilacji i konsolidacji programu CUDA.

1. Uaktywniamy konfigurację produkcyjną (*Release*) projektu. Czyścimy projekt (*Build — Clean Solution*). W razie potrzeby usuwamy ręcznie pozostałe w katalogu `x64/Release` pliki, za wyjątkiem `*.dll`. We właściwościach projektu włączamy flagę *Configuration Properties — CUDA C/C++ — Common — Keep Preprocessed Files* i ponownie budujemy projekt.
2. Oglądamy wszystkie pliki znajdujące się w katalogu `$(ProjectName)/$(ProjectName)/x64/Release`, po ich posortowaniu wg dat i godzin utworzenia. Jest ich mnóstwo! Zwracamy uwagę na najważniejsze z nich:

Kod dla host-a:

`kernel.cudafel.cpp`

kod źródłowy host-a po pierwszym przebiegu „front end”

Kod dla device:

`kernel.cudafel.gpu`

kod źródłowy device po pierwszym przebiegu „front end”

`kernel.cudafe2.gpu`

kod źródłowy device po drugim przebiegu „front end”

`kernel.ptx`

kod źródłowy device w assemblerze wirtualnego procesora GPU (wyszukujemy w nim ciąg znaków `.entry`)

`kernel.sm_30.cubin`

kod binarny device przetłumaczony dla domyślnej fizycznej architektury `sm_30` (tzn. `compute capability 3.0`)

`kernel.fatbin`

wzbogacony kod binarny device, mogący w ogólnym przypadku zawierać kod wygenerowany dla różnych architektur fizycznych i/lub źródło w języku PTX dla przyszłych, nieistniejących jeszcze platform sprzętowych

`kernel.fatbin.c`

plik `kernel.fatbin` w wersji tekstowej

Kod łączny dla host-a i device:

`kernel.cu.cpp.ii`

kod źródłowy hosta, zawierający także `kernel.fatbin.c`

`kernel.cu.obj`

plik zawierający kody binarne zarówno host-a, jak i device

`project.exe`

skonsolidowany plik binarny zawierający kody zarówno host-a, jak i device

1.1.12 Podglądanie rozkazów maszynowych

Jedną z metod oceny wydajności kodu naszej aplikacji jest analiza sposobu przetłumaczenia go przez kompilator na assembler.

1. Oglądamy końcówkę utworzonego w poprzednim punkcie pliku `.ptx`. Najważniejsze instrukcje „obliczeniowe” naszego programu to:

```
ld.global.f32
ld.global.f32
add.f32
```

Zauważmy, jak mały procent kodu jądra one stanowią!

2. Oglądanie kodu maszyny wirtualnej (`.ptx`) może być jednak dość mylące. Lepszy obraz sytuacji daje kod maszyny fizycznej. W katalogu `x64/Release` wykonujemy program:

```
cuobjdump -sass project.exe > kernel.sass
```

otrzymując kod źródłowy assemblera maszyny fizycznej. Identyfikujemy w nim instrukcje „obliczeniowe” naszego jądra. Sytuacja nie wygląda już tak tragicznie, jak na poziomie języka PTX.

3. Podobną analizę niskopoziomowego kodu można wykonać także za pomocą debuggera. W tym celu upewniamy się, że w menu *Tools — Options — Debugging* zaznaczone są obie opcje: *Enable Address Level Debugging* oraz *Show disassembly if source is not available*. Po zatrzymaniu się programu (najlepiej uruchamianego w konfiguracji *(Debug)*) na pałapce zastawionej w jądrze wybieramy *Debug — Windows — Disassembly*, a w otwartym okienku wybieramy *PTX and SASS*, aby zobaczyć zarówno rozkazy maszyny wirtualnej (PTX), jak i rzeczywistej (SASS).

1.1.13 Kompilacja z linii komendy

W ostatnim punkcie skorzystaliśmy z bardzo użytecznego narzędzia linii komendy, a mianowicie z programu `cuobjdump.exe`. Zestaw narzędzi CUDA oczywiście umożliwia także przeprowadzenie kompletnej kompilacji aplikacji z linii komendy.

1. Kompilacja programów CUDA z linii komendy powinna odbywać się w trybie 64-bitowym i wymaga odpowiedniego ustawienia zmiennych środowiskowych dla interpretera komend:

- Menu: *Start — Wszystkie programy — Visual Studio 2022 — x64 Native Tools Command Prompt*
- Wchodzimy (`cd`) do katalogu roboczego.

albo

- Uruchamiamy interpreter komend `cmd` i wchodzimy (`cd`) do katalogu roboczego.
- Wydajemy komendę:



```
"c:\Program Files\Microsoft Visual Studio\2022\Community\VC
\Auxiliary\Build\vcvarsall.bat" x64
```



2. Kompilujemy naszą aplikację programem `nvcc.exe`, będącym w zasadzie nakładką na kompilator host-a (Visual Studio 2022) i device (`nvopencc.exe` i `ptxas.exe`), konsolidator itp. itd.:

```
"c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\bin\
nvcc" -I"%NVCUDASAMPLES_ROOT%\common\inc" kernel.cu
```

a następnie wykonujemy go:

```
a.exe
```



3. Oczywiście można nadać programowi bardziej inteligentną nazwę. Służą do tego celu odpowiednie opcje linii komendy programu `nvcc.exe`. Przeanalizujmy te opcje w dokumencie "CUDA Compiler Driver `nvcc`" firmy NVIDIA (punkt 3.2), albo zapisując je do pliku tekstowego:

```
"c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\bin\
nvcc" --help > nvcc_opts.txt
```

i później oglądając ten plik za pomocą edytora.

1.1.14 Refaktoryzacja kodu

Kolejną czynnością będzie refaktoryzacja kodu naszej aplikacji na pliki `.c` lub `.cpp` dla host-a oraz pliki `.cu` dla device. Należy pomyśleć także o odpowiednim pliku nagłówkowym opisującym interfejs łączący obie części aplikacji. Po podziale kodu najpierw kompilujemy go z linii komendy, podając programowi `nvcc.exe` listę wszystkich modułów, a potem tworzymy odpowiedni projekt w środowisku Visual Studio 2022.

1.2 Program rysujący śliczne fraktale

1.2.1 Wersja CPU

Ten punkt należy zrealizować w domu podczas przygotowywania się do laboratorium. Utworzymy wersję CPU programu do rysowania słynnego fraktalu (zbioru Julii) i powiększania jego fragmentów. Do tworzenia grafiki wykorzystamy biblioteki OpenGL i GLUT². Przed-

²Kompilacja i uruchomienie przygotowywanego w domu programu wymaga zainstalowania jakiejś wersji biblioteki GLUT. W dalszej części niniejszego punktu zakłada się, że zainstalowane zostało środowisko CUDA w wersji 12.6, co pozwala na skorzystanie z następujących ścieżek do biblioteki freeglut (w wersji 64-bitowej):

- ścieżka do plików nagłówkowych, potrzebnych do kompilacji: `%NVCUDASAMPLES_ROOT%\common\inc`,
- ścieżka do plików opisu bibliotek, potrzebnych do konsolidacji: `%NVCUDASAMPLES_ROOT%\common\lib\x64`,
- ścieżka do plików bibliotek dynamicznych, potrzebnych do wykonania programu: `%NVCUDASAMPLES_ROOT%\bin\win64\Release` (zamiast ustawiać tę ścieżkę, można


stawione poniżej fragmenty kodu źródłowego są napisane bardzo minimalistycznie, aby nie zaciemniać struktury programu.

1. Przed napisaniem programu należy zapoznać się z literaturą obowiązkową:

- H.-O. Peitgen, H. Juergens, D. Saupe: *Granice chaosu. Fraktale. Część 2*, Wydawnictwo Naukowe PWN, Warszawa, 2002, p. 13.4 i rys. 13.26.

(dla ułatwienia na prywatnej stronie internetowej przedmiotu RIM zamieszczono plik .pdf zawierający zeskanowane odpowiednie strony) oraz ew. z literaturą uzupełniającą:

- J. Ganczarski: *OpenGL w praktyce*, BTC, Legionowo, 2008, Dodatek A.

2. Tworzymy plik `julia.c` (plik ten można również ściągnąć z prywatnej strony internetowej przedmiotu RIM) o następującej zawartości: 

```
#define DIM 1000 /* rozmiar rysunku w pikselach */

int julia(float x, float y)
{
    /* Tu trzeba będzie wstawić prawdziwy kod... */
    return 1;
}

void kernel(unsigned char *ptr,
            const int xw, const int yw,
            const float dx, const float dy,
            const float scale)
{
    /* przeliczenie współrzędnych pikselowych (xw, yw)
       na matematyczne (x, y) z uwzględnieniem skali
       (scale) i matematycznego środka (dx, dy) */
    float x = scale*(float)(xw-DIM/2)/(DIM/2) + dx,
          y = scale*(float)(yw-DIM/2)/(DIM/2) + dy;
    int offset /* w buforze pikseli */ = xw + yw*DIM;
    /* kolor: czarny dla uciekinierów (julia == 0)
       czerwony dla więźniów (julia == 1) */
    ptr[offset*4 + 0 /* R */] = (unsigned char) (255*julia(x,y));
    ptr[offset*4 + 1 /* G */] = 0;
    ptr[offset*4 + 2 /* B */] = 0;
    ptr[offset*4 + 3 /* A */] = 255;
}

/*****/

#define WIN32
```

także skopiować potrzebne biblioteki dynamiczne do katalogu projektu zawierającego pliki wykonywalne).

W przypadku innej instalacji użytkownik powinien zmodyfikować odpowiednio do swojej konfiguracji poniższą procedurę kompilacji, konsolidacji i wykonania programu.

```

#include "GL/freeglut.h"

static unsigned char pixbuf[DIM * DIM * 4];
static float dx = 0.0f, dy = 0.0f;
static float scale = 1.5f;

static void disp(void)
{
    glDrawPixels(DIM, DIM, GL_RGBA, GL_UNSIGNED_BYTE, pixbuf);
    glutSwapBuffers();
}

static void recompute(void)
{
    int xw, yw;
    for (yw = 0; yw < DIM; yw++)
        for (xw = 0; xw < DIM; xw++)
            kernel(pixbuf, xw, yw, dx, dy, scale);
    glutPostRedisplay();
}

static void kbd(unsigned char key, int x, int y)
{
    switch (key)
    {
        case 'p': dx += scale*(float)(x-DIM/2)/(DIM/2);
                  dy -= scale*(float)(y-DIM/2)/(DIM/2);
                  break;
        case 'z': scale *= 0.80f; break;
        case 'Z': scale *= 1.25f; break;
        case '=': scale = 1.50f; dx = dy = 0.0f; break;
        case 27: /* Esc */ exit(0);
    }
    recompute();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv); /* inicjacja biblioteki GLUT */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA); /* opcje */
    glutInitWindowSize(DIM, DIM); /* rozmiar okna graficznego */
    glutCreateWindow("RIM - fraktal Julii"); /* tytuł okna */
    glutDisplayFunc(disp); /* funkcja zwrotna zobrazowania */
    glutKeyboardFunc(kbd); /* funkcja zwrotna klawiatury */
    recompute(); /* obliczenie pierwszego rysunku */
    glutMainLoop(); /* główna pętla obsługi zdarzeń */
}

```

Plik ten nie jest jeszcze kompletnym źródłem, zaraz go uzupełnimy. Jego podstawowym

elementem jest funkcja `recompute`, wywoływana pierwszy raz przed główną pętlą zdażeń biblioteki OpenGL, a potem po każdej zmianie parametrów zobrazowania. Parametry zmieniają się za pomocą klawiatury, obsługiwanej przez funkcję zwrotną (callback) `kbd`. Zapewnia ona następujące sterowania:

`p` punkt wskazany myszą staje się nowym środkiem rysunku

`z` powiększenie rysunku

`Z` zmniejszenie rysunku

`=` przywrócenie domyślnego powiększenia i środka zobrazowania

`Esc` wyjście z programu

Z kolei funkcja `kernel` jest wywoływana przez funkcję `recompute` dla każdego piksela obrazu w zagnieżdżonej parze pętli. Funkcja `kernel` przelicza współrzędne pikselowe (x_w, y_w) na współrzędne matematyczne (x, y) z uwzględnieniem współczynnika skali i współrzędnych matematycznych środka rysunku. Punkty odpowiadające uciekinierom są zapisywane do bufora pikseli w kolorze czarnym, a odpowiadające więźniom – w kolorze czerwonym. Zero-jedynkową informację o tym, czy punkt o współrzędnych matematycznych (x, y) (odpowiadających liczbie zespolonej $z = x + jy$) jest uciekinierem (0), czy więźniem (1), zwraca funkcja `julia` o prototypie:

```
int julia(float x, float y);
```

Na razie, jak widać, funkcja ta jest pusta – zakłada, że wszyscy są więźniami.

3. Uzupełniamy w pliku `julia.c` funkcję `julia`, na podstawie pseudokodu zamieszczonego na samym końcu p. 13.4 pierwszej pozycji literatury. Jako parametr c przekształcenia $z_{n+1} = z_n^2 + c$ wybieramy $c = -0.123 + j0.745$, co odpowiada tzw. „królikowi Douady’ego”. Maksymalną liczbę iteracji (parametr k wspomnianego pseudokodu) ustalamy na razie w granicach 100...1000, w następnym punkcie nieco z nim poeksperymentujemy.
4. Kompilujemy program albo tworząc odpowiedni projekt zintegrowanego środowiska Microsoft Visual Studio 2022, albo posługując się linią komendy najpierw w celu ustawienia zmiennych środowiskowych, a następnie w celu przeprowadzenia właściwej kompilacji i konsolidacji:

```
"c:\Program Files\Microsoft Visual Studio\2022\Community\VC\
Auxiliary\Build\vcvarsall.bat" x64
cl julia.c /Fejulia_cpu.exe /W4 /I"%NVCUDASAMPLES_ROOT%/common/
inc" /link /LIBPATH:"%NVCUDASAMPLES_ROOT%/common/lib/x64"
```

Uruchamiamy program, ustawiając uprzednio ścieżkę do biblioteki GLUT:

```
set PATH=%PATH%;%NVCUDASAMPLES_ROOT%\bin\win64\Release
julia_cpu.exe
```

Powinniśmy uzyskać rysunek podobny do pokazanego na rys. 13.26 pierwszej pozycji literatury (tylko z wypełnionym na czerwono wnętrzem).



5. Spróbujmy pobadać fraktal Julii, powiększając jego wybrane fragmenty. Czy starczy nam cierpliwości, aby osiągnąć tak duże powiększenie, że obraz zacznie cierpieć na „pikselozę”?

1.2.2 Wersja GPU

W laboratorium napiszemy i uruchomimy wersję GPU programu do rysowania zbioru Julii.

1. Kopiujemy plik `julia.c` na nazwę `julia.cu`. Na samym początku funkcji `main` wywołujemy funkcję `cudaSetDevice` (oczywiście ze sprawdzaniem ew. błędów makrem `checkCudaErrors`).

2. Dołączamy plik nagłówkowy:

```
#include "helper_cuda.h"
```

3. Funkcję `julia` deklarujemy jako `__device__`.

4. funkcję `kernel` deklarujemy jako `__global__`. Usuwamy z listy jej parametrów formalnych współrzędne pikselowe `xw` i `yw`. Zamiast tego deklarujemy zmienne automatyczne o takich samych nazwach i inicjujemy je na podstawie indeksów bloku i wątku. Pamiętajmy o zabezpieczeniu przed zbyt dużymi wartościami współrzędnych `xw` i `yw` (zetknęliśmy się już z tym problemem w punkcie 1.1.4).

5. Deklarujemy wskaźnik do dodatkowego bufora pikseli w pamięci device:

```
static unsigned char *d_pixbuf;
```

Na początku funkcji `main`, tuż po wywołaniu `cudaSetDevice`, alokujemy pamięć urządzenia o takim samym rozmiarze, jak rozmiar bufora `pixbuf` i przypisujemy tę pamięć do wskaźnika `d_pixbuf`. Dla uproszczenia nie będziemy się kłopotali zwalnianiem tej pamięci przy wyjściu z programu.

6. W funkcji `recompute` zamiast dwóch zagnieżdżonych pętli `for` umieszczamy wywołanie jądra

```
dim3 dimGrid((DIM + DIM_BLOCK - 1) / DIM_BLOCK,
              (DIM + DIM_BLOCK - 1) / DIM_BLOCK);
dim3 dimBlock(DIM_BLOCK, DIM_BLOCK);
kernel<<<dimGrid, dimBlock>>>(d_pixbuf, dx, dy, scale);
```

gdzie `DIM_BLOCK` jest odpowiednio zdefiniowaną stałą:

```
#define DIM_BLOCK 16
```

Po wywołaniu jądra oczywiście sprawdzamy status tej operacji (jak i każdej innej operacji CUDA), a następnie kopiujemy zawartość bufora pikselowego urządzenia (`d_pixbuf`) do bufora pikselowego komputera nadrzędnego (`pixbuf`).

7. Kompilujemy program albo tworząc odpowiedni projekt zintegrowanego środowiska Microsoft Visual Studio 2022, podobnie jak w punkcie 1.1.1:

- Menu: *Start* — *Wszystkie programy* — *Visual Studio 2022*.
- Menu: *File* — *New* — *Project*, w zakładce *Installed* rozwijamy *NVIDIA* i wybieramy pozycję *CUDA* — *CUDA 12.6 Runtime*.
- Usuwamy z projektu (*Remove* — *Delete* w menu kontekstowym) automatycznie utworzony plik `kernel.cu`.
- Dołączamy do projektu (*Add* — *Existing Item* w menu kontekstowym) nasz plik `julia.cu`.
- Ustawiamy konfigurację (*Debug/Release*), a następnie platformę *x64*.
- W polu *Solution Explorer* klikamy prawym guzikiem myszy na nazwę projektu i wybieramy *Properties* (ostatnią pozycję). Do opcji *Configuration Properties* — *VC++ Directories* — *Executable Directories* dodajemy pozycję `$(NVCUDASAMPLES_ROOT)/bin/win64/Release` (dla konfiguracji produkcyjnej projektu).
- W tym samym okienku właściwości projektu do opcji *Configuration Properties* — *VC++ Directories* — *Include Directories* dodajemy pozycję `$(NVCUDASAMPLES_ROOT)/common/inc` dla wszystkich konfiguracji projektu.
- Po raz kolejny w tym samym okienku właściwości projektu do opcji *Configuration Properties* — *VC++ Directories* — *Library Directories* dodajemy pozycję `$(NVCUDASAMPLES_ROOT)/common/lib/x64` dla wszystkich konfiguracji projektu.
- Raz jeszcze w tym samym okienku właściwości projektu do opcji *Configuration Properties* — *Linker* — *Input* — *Additional Dependencies* dodajemy pozycję `freeglut.lib` dla wszystkich konfiguracji projektu.
- Do podkatalogu odpowiadającego aktualnej konfiguracji projektu *Release/Debug* wgrywamy plik `freeglut.dll` z katalogu `$(NVCUDASAMPLES_ROOT)/bin/win64/Release` (lub odpowiednio `.../Debug`).

albo posługując się linią komendy najpierw w celu ustawienia zmiennych środowiskowych, a następnie w celu przeprowadzenia właściwej kompilacji i konsolidacji:

```
"c:\Program Files\Microsoft Visual Studio\2022\Community\VC\
Auxiliary\Build\vcvarsall.bat" x64
"c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\bin\
nvcc" julia.cu -o julia_gpu.exe -I"%NVCUDASAMPLES_ROOT%\
common/inc" -L"%NVCUDASAMPLES_ROOT%\common/lib/x64"
```

Uruchamiamy program, ustawiając uprzednio ścieżkę do biblioteki GLUT:

```
set PATH=%PATH%;%NVCUDASAMPLES_ROOT%\bin\win64\Release
julia_gpu.exe
```

Powinniśmy uzyskać rysunek identyczny jak dla programu wykonywanego przez CPU.

8. Spróbujmy pobadać fraktal Julii, powiększając jego wybrane fragmenty. Tym razem powinniśmy mieć odczucie wyraźnie szybszego reagowania komputera na polecenia wydawane z klawiatury. Bez problemu można dojść do tak dużego powiększenia, że obraz zacznie wykazywać „pikselozę”.



9. Spróbujmy potwierdzić to wrażenie większej reaktywności obiektywnym pomiarem. Można (podobnie jak w punkcie 1.1.5) zmierzyć czas wykonania funkcji `recompute` w milisekundach i wyświetlić go w tytule okna funkcją `glutSetWindowTitle(char *name)`.
10. Trochę eksperymentujemy z maksymalną liczbą iteracji pętli **for** (badającej, czy dany punkt jest uciekinierem, czy więźniem), próbując ją zmniejszyć (np. do 10) lub zwiększyć (np. do 10000).
11. **if** wersja GPU jest wyraźnie szybsza niż wersja CPU **then**
 OK, przecież właśnie o to nam chodziło.
else
 Bez paniki, na następnym laboratorium poznamy kilka technik optymalizacji.
end if

Rozdział 2

Optymalizacja programów w języku CUDA C

Opis niniejszego ćwiczenia będzie zakładał, że Czytelnik posiadał już (np. podczas pierwszego laboratorium) podstawowe umiejętności dotyczące posługiwania się środowiskiem CUDA. Skrypt nie będzie zatem opisywał *jak* należy wykonać poszczególne czynności, tylko skupi się na tym, *co* należy wykonać. W razie potrzeby zaleca się sięgnięcie po odpowiedzi do poprzedniego rozdziału.

Podczas laboratorium trzeba notować czasy działania i wydajności wszystkich implementowanych wariantów programów, równocześnie „chwaląc się” prowadzącemu każdym wykonanym zadaniem i możliwie szybko zgłaszając ew. trudności w jego wykonaniu. Nie będziemy w ramach ćwiczenia liczyli przyspieszenia kodu względem implementacji na CPU, tylko jednej implementacji GPU względem innej.

W trakcie implementowania różnych wariantów kodu często powstaje konieczność chwilowego powrotu do jego starej wersji. Jednym z wygodnych sposobów na poradzenie sobie z problemem zarządzania kilkoma wersjami modułu (lepszym niż często stosowane przez studentów komentowanie dużych fragmentów kodu wewnątrz plików) jest dołączenie do projektu zintegrowanego środowiska Microsoft Visual Studio 2022 wszystkich wersji pliku źródłowego i zastosowanie mechanizmu wykluczania ich z procesu budowania kodu binarnego. W tym celu we właściwościach danego pliku należy odpowiednio ustawić *Properties* — *Configuration Properties* — *General* — *Excluded From Build* — *Yes/No*.

Trzeba też pamiętać, żeby zawsze przed pomiarami efektywności każdego kolejnego wariantu programu upewnić się co do jego poprawności, zwracając uwagę na wyniki testu zgodności wyników ze wzorcem oraz uruchamiając program za pomocą wykorzystywanego już w poprzednim ćwiczeniu narzędzia `cuda-memcheck`.

Przed laboratorium należy wykonać w domu zadania [2.1.1](#), [2.1.2](#) oraz [2.1.3](#). Uwaga: w laboratorium jest dostępny Internet, więc dane i ew. kod tego zadania można przynieść z domu albo na nośniku USB, albo zapisać na swoim koncie na jakimś, najlepiej wydziałowym, serwerze.

2.1 Filtracja zakłóceń sygnału akustycznego

W tym punkcie będziemy filtrowali sygnał akustyczny (audio) filtrem cyfrowym o skończonej odpowiedzi impulsowej (FIR – *finite impulse response*). Projekt będzie się więc nazywał

audiofir (nie mylić z audiofilem).

2.1.1 Współczynnik CGMA dla filtracji FIR

Należy oszacować współczynnik pamięciowy złożoności obliczeniowej CGMA (*Compute to Global Memory Access ratio*) dla filtracji dyskretnego sygnału rzeczywistego y_{in} o długości L próbek filtrem o skończonej odpowiedzi impulsowej h rzędu n . Wskazówka dla osób spoza sygnałowego nurtu kształcenia: wynikiem filtracji jest też sygnał rzeczywisty y_{out} o długości L próbek, a każda próbka wyjściowa jest iloczynem skalarnym wektora $n + 1$ próbek sygnału wejściowego i wektora rzeczywistych współczynników filtru h o rozmiarze $n + 1$:

$$y_{out}[i] = [y_{in}[i], y_{in}[i - 1], \dots, y_{in}[i - n]] \cdot [h_0, h_1, \dots, h_n]^T = \quad (2.1)$$

$$= \sum_{k=0}^n y_{in}[i - k] h_k, \quad (2.2)$$

dla $i = 0, \dots, L - 1$. Przyjmuje się przy tym, że:

$$y_{in}[i] = 0 \quad \text{dla} \quad i < 0. \quad (2.3)$$

2.1.2 Implementacja filtracji FIR na CPU

Ten punkt należy zrealizować w domu podczas przygotowywania się do laboratorium. Utworzymy „szkolną” wersję programu filtracji FIR na CPU.



- Tworzymy plik `audiofir0.c` o następującej zawartości:

```
#include "audiofir.h"

static void audiofir_kernel(int i,
    float *yout, float *yin, float *coeff, int n, int len)
{
    /* Tu trzeba będzie wstawić prawdziwy kod... */
}

void audiofir(
    float *yout, float *yin, float *coeff, int n, int len, ...)
{
    int i;
    for (i = 0; i < len; i++)
        audiofir_kernel(i, yout, yin, coeff, n, len);
    for (i = 0; i < len; i++)
        audiofir_kernel(i, yout+len, yin+len, coeff, n, len);
}
```



Plik ten nie jest jeszcze kompletnym źródłem, zaraz go uzupełnimy. Jego podstawowym elementem jest funkcja `audiofir`. Jej prototyp znajduje się w pliku `audiofir.h` o następującej zawartości:

```

#if defined __cplusplus
extern "C"
#endif
void audiofir(
    float *yout, float *yin, float *coeff, int n, int len, ...);

```

Funkcja `audiofir` dostaje jako parametry wejściowe próbki sygnału wejściowego do filtracji `yin` (najpierw `len` próbek kanału lewego, potem `len` próbek kanału prawego) oraz współczynniki filtru FIR `coeff` (wektor $n + 1$ współczynników). Funkcja ta dwukrotnie (raz dla lewego i raz dla prawego kanału stereofonicznego) przebiega po próbkach danych wejściowych `yin` dla każdej (i -tej) z nich wywołując zasadnicze jądro filtracji `audiofir_kernel` obliczające iloczyn skalarny (2.2).

- Uzupełniamy w pliku `audiofir0.c` funkcję `audiofir_kernel` zgodnie ze wzorem (2.2). Warto już w tym momencie napisać ten kawałek kodu „porządnie”, tzn. wewnątrz pętli sumowania po k kolejne iloczyny należy dosumowywać *nie* do elementu tablicy wyjściowej odpowiadającej sygnałowi y_{out} , tylko do oddzielnie zadeklarowanej skalarnej zmiennej automatycznej, którą dopiero na zewnątrz pętli po zakończeniu sumowania podstawimy na odpowiedni element tablicy wyjściowej. W ten sposób znacząco zmniejszymy liczbę odwołań kodu do pamięci. Pamiętajmy także, aby wewnątrz pętli obliczania iloczynu skalarnego dodać zabezpieczenie `if (i >= k)` przed sięganiem przed pierwszą próbkę sygnału $y_{in}[0]$, czyli do elementów wektora `yin` o ujemnych indeksach, zgodnie z założeniem wynikającym ze wzoru (2.3). Parametr `len` funkcji `audiofir_kernel` nie jest na razie wykorzystywany, ale przyda się, gdy przejdziemy na implementację na procesorze GPU.
- Plik `audiofir0.c` zawiera część „wykonawczą” naszej aplikacji. Oprogramowanie szkieletowe (w tym funkcja `main`) zawarte jest w pliku `audiofir.c` o następującej zawartości:

```

/* Filtracja sygnału AUDIO filtrem FIR - szkielet */

#ifndef _MSC_VER
# define _CRT_SECURE_NO_WARNINGS
#endif

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "audiofir.h"

/*****/

void alloc_mem(int n, int len,
               float **coeff_ptr, float **yin_ptr,
               float **yref_ptr, float **yout_ptr)
{

```



```

*coeff_ptr = (float *) malloc((1 + n) * sizeof(float));
* yin_ptr = (float *) malloc(2 * len * sizeof(float));
* yref_ptr = (float *) malloc(2 * len * sizeof(float));
* yout_ptr = (float *) malloc(2 * len * sizeof(float));
}

void free_mem(float *coeff_ptr, float *yin_ptr,
              float *yref_ptr, float *yout_ptr)
{
    free(coeff_ptr);
    free( yin_ptr);
    free( yref_ptr);
    free( yout_ptr);
}

void read_data(int *n_ptr, int *len_ptr,
               float **coeff_ptr, float **yin_ptr,
               float **yref_ptr, float **yout_ptr)
{
    FILE *f = fopen("audiofir_in.dat", "rb");

    fread( n_ptr, sizeof(int), 1, f);
    fread(len_ptr, sizeof(int), 1, f);

    alloc_mem(*n_ptr, *len_ptr,
              coeff_ptr, yin_ptr, yref_ptr, yout_ptr);

    fread(*coeff_ptr, sizeof(float), 1 + *n_ptr, f);
    fread(* yin_ptr, sizeof(float), 2 * *len_ptr, f);
    fread(* yref_ptr, sizeof(float), 2 * *len_ptr, f);

    fclose(f);
}

void write_data(int len, float *y)
{
    FILE *f = fopen("audiofir_out.dat", "wb");
    fwrite(y, sizeof(float), 2 * len, f);
    fclose(f);
}

/*****

void audiocmp(float *yout, float *yref, int len)
{
    int k;
    float d, e = -1.0f;
    for (k = 0; k < 2 * len; k++)
        if ((d = fabsf(yout[k] - yref[k])) > e)

```



```

        e = d;
        printf("max. abs. err. = %.1e\n", e);
    }

/*****

#ifdef _WIN32

#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,
                  double 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);
}

#else

#include <time.h> /* requires linking with rt library
                  (command line option -lrt) */

typedef struct timespec app_timer_t;

#define timer(t_ptr) clock_gettime(CLOCK_MONOTONIC, t_ptr)

void elapsed_time(app_timer_t start, app_timer_t stop)
{
    double etime;
    etime = 1e+3 * (stop.tv_sec - start.tv_sec) +
            1e-6 * (stop.tv_nsec - start.tv_nsec);
    printf("CPU (total!) time = %.3f ms (%6.3f GFLOP/s)\n",
           etime, 1e-6 * flop / etime);
}

#endif

*****/

```

```

int main(int argc, char *argv[])
{
    app_timer_t start, stop;
    int n, len;
    float *coeff, *yin, *yref, *yout;
    read_data(&n, &len, &coeff, &yin, &yref, &yout);
    timer(&start);
    audiofir(yout, yin, coeff, n, len, argc, argv);
    timer(&stop);
    elapsed_time(start, stop,
        2*((double)n+1) * 2*((double)len));
    audiocmp(yout, yref, len);
    write_data(len, yout);
    free_mem(coeff, yin, yref, yout);
    if (IsDebuggerPresent()) getchar();
    return 0;
}

```

Kod ten jest napisany bardzo minimalistycznie, aby nie zaciemniać struktury programu. Część szkieletowa kodu nie będzie ulegała zmianie nawet po przejściu na implementację programu na procesorze graficznym – tworzyć będziemy tylko kolejne warianty `audiofirx.cu` kodu „wykonawczego”.

- Kompilujemy program albo tworząc odpowiedni projekt zintegrowanego środowiska Microsoft Visual Studio 2022, albo posługując się linią komendy. W obu przypadkach pamiętajmy, aby był to projekt 64-bitowy. Projekt zawiera dwa pliki źródłowe: `audiofir.c` i `audiofir0.c` oraz jeden plik nagłówkowy `audiofir.h`. Nie są wymagane żadne dodatkowe biblioteki. Warto pamiętać o włączeniu wysokiego poziomu ostrzeżeń (opcja `/W4` kompilatora `cl`).
- Zaczniemy od uruchomienia programu w prostej konfiguracji testowej. Do celów uruchamiania dogodnie jest pracować na całkowicie sztucznym sygnale i filtrze, które będą w debuggerze pokazywały jak na dłoni, co naprawdę się dzieje w programie. Do wygenerowania testowego pliku `audiofir_in.dat` służy program MATLAB-owski¹ `testfir.m` o następującej zawartości:

```

function [yorg, yin, yref, Fs] = testfir

%% generacja oryginalnego sygnału
Fs = 44100;
yorg = [1 2 3 4 5 6 7 8 9];
yorg = [yorg.', -yorg.'];
len = length(yorg);

%% brak zakłócenia sygnału szumem i przydźwiękiem sieciowym
yin = yorg;

```

¹Program ten można także uruchomić w darmowym środowisku GNU Octave, co pozwala na wykonanie niniejszego punktu w domu.

```

%% zaprojektowanie filtru i filtracja
coeff = [1e0 1e1 1e2 1e3];
n = length(coeff) - 1;

yref = filter(coeff, 1, yin);

%% zapamiętanie filtru i sygnałów w pliku binarnym
f = fopen('audiofir_in.dat', 'wb');
fwrite(f, [n, len], 'int');
fwrite(f, coeff, 'single');
fwrite(f, yin, 'single');
fwrite(f, yref, 'single');
fclose(f);

```

Oczywiście odpowiednio modyfikując ten plik, można go dostosować do własnych potrzeb – jest on raczej ilustracją pewnego pomysłu na uruchamianie programu. Jeśli program działa prawidłowo na tak prostym sygnale (daje zerowy błąd wektora `yout` względem `yref`), to przechodzimy do następnego punktu.

2.1.3 Wybór sygnału akustycznego

Wyszukajmy w swojej domowej fonotece jakieś nasze ulubione nagranie MP3 spełniające następujące warunki:

1. Jest to nagranie stereofoniczne (ma ani mniej, ani więcej, tylko dokładnie dwa kanały).
2. Nagranie ma częstotliwość próbkowania 44100 Hz (tak jak na płycie CD).
3. Nagranie nie trwa zbyt krótko, **ani zbyt długo** (najlepiej 1–2 minuty; przetwarzanie dłuższych nagrań w trybie uruchomieniowym *Debug* trwa zbyt długo).
4. Muzyka jest dość spokojna (kameralna klasyczna, ballada itp.), ale ma sporo wysokich tonów.

Uwaga: warto przynieść na laboratorium (oczywiście oprócz wyszukanego nagrania...) własne słuchawki dobrej jakości z wtykiem tzw. mini-jack (typowy wtyk do wyjścia karty dźwiękowej komputera), o impedancji ok. 30 Ω .

2.1.4 Niszczanie i odtwarzanie sygnału akustycznego

Za pomocą załączonego programu MATLAB-owskiego `audiofir.m`: o następującej zawartości:



```

function [yorg, yin, yref, Fs] = audiofir(file)

%% odczyt oryginalnego sygnału
[yorg, Fs] = audioread(file); % yorg = [y_left, y_right]
len = length(yorg);
assert(Fs == 44100);
assert(size(yorg, 2) == 2);

```

```

%% zakłócenie sygnału szumem i przydźwiękiem sieciowym
noise = filter(firl(128, 11000/(Fs/2), 'high',...
               blackmanharris(129)), 1,...
               randn(size(yorg))); % szum w.cz.

hum = sin(2*pi*50*(0:len-1).'/Fs); % przydźwięk 50Hz
yin = yorg + noise + [hum, hum];

%% zaprojektowanie filtru i odfiltrowanie zakłóceń
n = 1024;
coeff = firpm(n, [0 70 90 9900 10100 (Fs/2)]/(Fs/2),...
              [0 0 1 1 0 0 ],...
              [10.1606 1 10000 1]);

tic
yref = filter(coeff, 1, yin);
toc

%% zapamiętanie filtru i sygnałów w pliku binarnym
f = fopen('audiofir_in.dat', 'wb');
fwrite(f, [n, len], 'int');
fwrite(f, coeff, 'single');
fwrite(f, yin, 'single');
fwrite(f, yref, 'single');
fclose(f);

```

- Odczytujemy z dysku oryginalny sygnał y oraz jego częstotliwość próbkowania F_s i liczbę próbek $\text{len} = L$.
- Dodajemy do sygnału silny szum wysokoczęstotliwościowy (w pasmie powyżej 10100 Hz) i przydźwięk sieci energetycznej 50 Hz, otrzymując sygnał y_{in} .
- Filtrujemy zakłócony sygnał filtrem FIR wysokiego rzędu $n = 1024$, otrzymując sygnał y_{ref} stanowiący punkt odniesienia do sprawdzania poprawności naszych własnych procedur filtracji. Filtr prawie całkowicie wycina pasmo powyżej 9900 Hz (a więc szum) i ma tzw. zero charakterystyki amplitudowej (a więc w ogóle nie przepuszcza sygnału) na częstotliwości 50 Hz.
- Przy okazji filtracji mierzymy i notujemy czas jej realizacji przez MATLAB-owską funkcję `filter`. Jest to wysoko zoptymalizowana funkcja zakodowana w assemblerze.
- Zapamiętujemy w pliku dyskowym `audiofir_in.dat` rząd filtru i długość sygnału jako liczby całkowite, a potem współczynniki filtru `coeff` oraz sygnał zakłócony y_{in} i sygnał po filtracji y_{uref} jako wektory liczb zmiennoprzecinkowych pojedynczej precyzji.

W tym celu wykonujemy w okienku komend MATLAB-a polecenie:

```
[yorg, yin, yref, Fs] = audiofir('moje_ulubione_nagranie.mp3');
```

Każdy z trzech otrzymanych sygnałów jest macierzą, której pierwsza kolumna odpowiada lewemu, a druga – prawemu kanałowi stereo.

Odsłuchujemy poszczególne nagrania poleceniami:

```
soundsc(yorg, Fs);
soundsc(yin, Fs);
soundsc(yref, Fs);
```

(Odtwarzanie można w każdej chwili przerwać naciskając kombinację klawiszy *Ctrl-C*.) Jaka jest subiektywna („na ucho”) jakość sygnału po filtracji?

2.1.5 Uruchomienie filtracji FIR na CPU dla sygnału akustycznego

1. Uruchamiamy program w konfiguracji uruchomieniowej, po czym weryfikujemy jego poprawność, sprawdzając wartość maksymalnego bezwzględnego błędu drukowaną przez program. Powinna być ona mała – powiedzmy rzędu 10^{-6} . Jeśli tak jest, to kompilujemy program w konfiguracji produkcyjnej (z włączoną optymalizacją – opcja */Ox* kompilatora *cl*) i notujemy czas działania programu oraz osiągniętą wydajność w GFLOP/s, a następnie porównujemy czas działania z filtracją MATLAB-owską (przypomnijmy – wysoce zoptymalizowaną!). Jeśli program daje błędne wyniki (maksymalny bezwzględny błąd jest za duży, np. rzędu 1), to trzeba go jednak porządnie uruchomić.

2.1.6 Pierwsza (naiwna) implementacja w środowisku CUDA

W laboratorium napiszemy i uruchomimy wersję GPU programu do filtracji FIR. Zaczniemy od zakodowania w głównej funkcji wykonawczej *audiofir* typowych siedmiu kroków przetwarzania CUDA.

1. Kopiujemy plik *audiofir0.c* na nazwę *audiofir1.cu*. Dołączamy do niego plik nagłówkowy:

```
#include "helper_cuda.h"
```

i dodajemy do opcji *Configuration Properties — VC++ Directories — Include Directories* projektu pozycję $\$(NVCUDASAMPLES_ROOT)/common/inc$ niezbędną do jego znalezienia. Na samym początku funkcji *audiofir* wywołujemy funkcję *cudaSetDevice* (oczywiście wszystkie wywołania funkcji *cudaXxxx* opatrujemy sprawdzaniem ew. błędów makrem *checkCudaErrors*).

2. Deklarujemy trzy wskaźniki typu **float** * i alokujemy za pomocą funkcji *cudaMalloc* w pamięci urządzenia trzy wektory: współczynników filtru ($n + 1$ elementów) oraz sygnału wejściowego i wyjściowego filtru (oba po $2 \cdot len$ elementów – na kanał lewy i prawy).
3. Kopiujemy za pomocą funkcji *cudaMemcpy* do pamięci urządzenia wektor współczynników filtru *coeff* i sygnał wejściowy *yin*.
4. Zamiast dwóch pętli **for** (dla kanału lewego i prawego) wpisujemy dwa wywołania funkcji jądra *audiofir_kernel*. Musimy w tym momencie zdecydować o konfiguracji wykonania naszego programu. Naturalne wydaje się związanie pojedynczego wątku

z każdą próbką przefiltrowanego sygnału, czyli elementem wyjściowego wektora `yout`. Ponieważ sygnał w każdym kanale stereo jest jednowymiarowy (funkcja czasu), więc i konfiguracja sieci bloków wątków oraz każdego bloku będzie jednowymiarowa. Założymy, że każdy blok będzie obejmował `K` wątków, przy czym wielkość tę zdefiniujemy na początku pliku `audiofir1.cu`:

```
#define K 512
```

W wyniku tych decyzji konfiguracja każdego z dwóch wykonanń jądra przyjmie postać:

```
<<<(len+K-1)/K, K>>>
```

Oczywiście na liście parametrów aktualnych funkcji jądra `audiofir_kernel` musimy pominąć indeks `i` oraz pamiętać równocześnie o podaniu w jej argumentach wskaźników do wektorów w pamięci urządzenia, a nie gospodarza. Po wywołaniu obu jąder wstawiamy sprawdzanie statusu tych operacji w postaci:

```
checkCudaErrors(cudaGetLastError());
```

Dla celów pomiaru czasu wykonania *samego* jądra, *przed* pierwsze z jego wywołań wpisujemy kod:

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

a po wywołaniu funkcji `cudaGetLastError` – kod:

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

5. Czekamy na zakończenie wykonywania funkcji jądra:

```
checkCudaErrors(cudaDeviceSynchronize());
```

6. Kopiujemy wyniki pracy jądra (przefiltrowany sygnał) do pamięci gospodarza za pomocą funkcji `cudaMemcpy`.
7. Zwalniamy przydzieloną pamięć urządzenia (`cudaFree`). Następnie dla potrzeb debugera wywołujemy funkcję `cudaDeviceReset` i drukujemy wyniki pomiaru czasu wykonania jądra:

```
printf("GPU (kernel) time = %.3f ms (%6.3f GFLOP/s)\n",
      elapsedTime,
      1e-6 * 2*((double)n+1) * 2*((double)len) /
      elapsedTime);
```

Warto zwrócić tu uwagę na konieczność rzutowania rozmiarów problemu na typ `double` aby uniknąć przepełnień przy operacjach stałoprzecinkowych.

Zmiany konieczne w funkcji jądra `audiofir_kernel` są już niewielkie – wystarczy poprzedzić jej deklarację modyfikatorem `__global__` oraz z listy parametrów formalnych usunąć indeks `i`. Zamiast tego musimy go odtworzyć na początku ciała funkcji na podstawie indeksów wątku w bloku i bloku w sieci, a także zabezpieczyć się przed sięganiem poza ostatnią próbkę sygnału (zwróćmy uwagę, że w konfiguracji wykonania liczba bloków została zaokrąglona w górę):

```
int i = threadIdx.x + blockIdx.x*blockDim.x; if (i < len) {
    ...
}
```

Kompilujemy i uruchamiamy program, notujemy czas wykonania funkcji jądra i narzuty na kopiowanie danych itp.

2.1.7 Lekko zmieniona implementacja w środowisku CUDA

W pierwszym kroku lekko uprościmy kod jądra, aby łatwiej nam było realizować kroki następne. Pozbędziemy się mianowicie obu sprawdzeń `if (i < len)` oraz `if (i >= k)`.

1. Kopiujemy plik `audiofir1.cu` na nazwę `audiofir2.cu`.
2. Aby można było pozbyć się sprawdzeń, poświęcimy nieco pamięci, odpowiednio dopełniając sygnał każdego z kanałów stereofonicznych na końcu zerami tak, aby jego długość była wielokrotnością K , a także poprzedzając sygnał w każdym kanale n zerami. Alo-kacja wektora sygnału wejściowego w pamięci urządzenia powinna zatem zamiast $2L$ elementów dotyczyć $2(L_1 + n)$ elementów, gdzie

$$L_1 = \left\lceil \frac{L + K - 1}{K} \right\rceil K, \quad (2.4)$$

zaś alokacja wektora sygnału wyjściowego – dotyczyć $2L_1$ elementów.

3. Zamiast kopiowania $2L$ elementów sygnału wejściowego filtru z pamięci gospodarza do pamięci urządzenia, musimy teraz (nie zmieniając układu danych w pamięci gospodarza, a tylko zmieniając sposób przesyłania danych do urządzenia):
 - wyzerować n pierwszych elementów (funkcją `cudaMemset`²),
 - skopiować L kolejnych elementów z wektora gospodarza (kanał lewy) (za pomocą funkcji `cudaMemcpy`),
 - przeskoczyć $L_1 - L$ następnych elementów (można je dla porządku także wyzerować, żeby nie znalazł się wśród nich jakiś NaN),
 - wyzerować n kolejnych elementów,

²Sposób jej wywołania odczytamy z dokumentacji środowiska CUDA: *Start — Wszystkie Programy — NVIDIA Corporation — CUDA Documentation*. W okienku przeglądarki, które się otworzy, w lewym panelu wybieramy *CUDA API References — CUDA Runtime API — Modules — Memory Management*, po czym w prawym panelu wyszukujemy łańcuch znaków `cudaMemset`.

- skopiować L kolejnych elementów z wektora gospodarza (kanał prawy),
- przeskoczyć $L_1 - L$ ostatnich elementów (można je dla porządku także wyzerować, żeby nie znalazł się wśród nich jakiś NaN),

Zaleca się tu zrobienie odpowiedniego rysunku „mapy pamięci” urządzenia. Należy zwrócić uwagę na to, że teraz dane kanału lewego i prawego (poprzedzone zerami) nie tworzą w pamięci urządzenia ciągłego obszaru. Oznacza to konieczność dokładnego przemyślenia arytmetyki adresowej przy przesyłaniu danych do urządzenia i przy zadawaniu parametrów funkcji jądra.

4. W kodzie funkcji jądra, tak jak już wspomniano powyżej, usuwamy oba sprawdzenia `if (i < len)` oraz `if (i >= k)`. W tym momencie można się też pozbyć z listy parametrów jądra długości sygnału `len`.

Nieco uprościliśmy kod jądra kosztem komplikacji kodu funkcji wywołującej to jądro. Nasze działania były raczej czysto porządkowe i nie powinny mieć istotnego wpływu na wydajność programu, co weryfikujemy przez jego kompilację i uruchomienie.

2.1.8 Umieszczenie współczynników filtru w pamięci stałej

Wprost narzucającym się sposobem optymalizacji naszego kodu jest umieszczenie współczynników filtru w pamięci stałej. Zauważmy bowiem, że współczynniki w kodzie jądra tylko czytamy³.

1. Kopiujemy plik `audiofir2.cu` na nazwę `audiofir3.cu`.
2. Na głównym poziomie tak utworzonego pliku deklarujemy maksymalny rząd filtru:

```
#define N 1024 // maksymalny rząd filtru FIR
```

a następnie deklarujemy położony w pamięci stałej wektor współczynników:

```
__constant__ static float fir_coeff[N + 1];
```

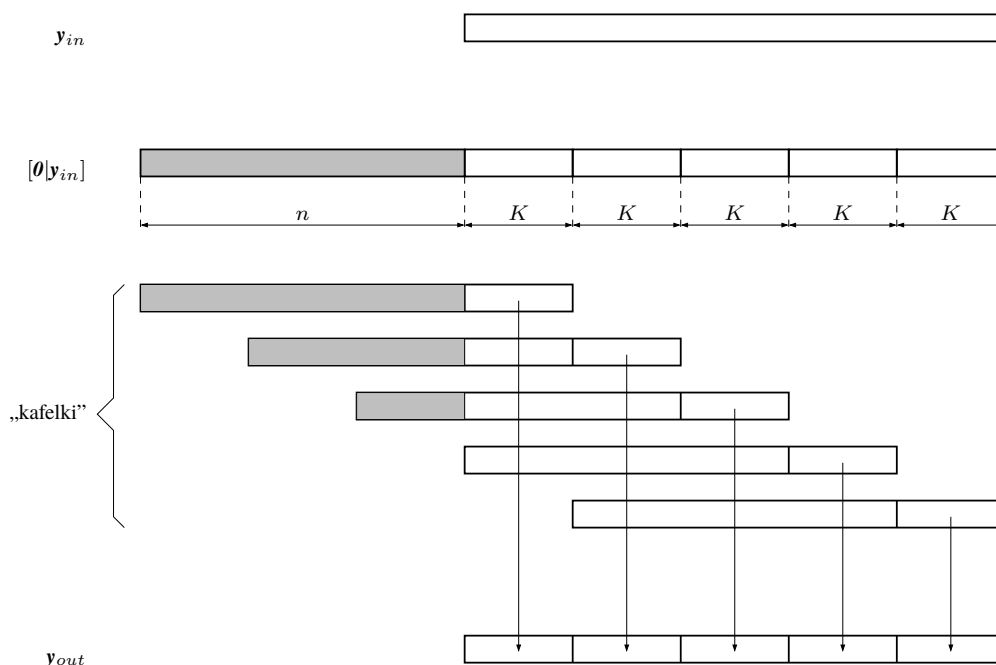
3. Z list parametrów formalnych jądra `audiofir_kernel` usuwamy wektor współczynników `'coeff'`, a nazwę tego wektora w ciele funkcji zmieniamy na `fir_coeff`.
4. W ciele funkcji `audiofir` sprawdzamy spełnienie warunku (`n <= N`) (instrukcją `if` lub funkcją `assert`), a następnie usuwamy wszelkie odniesienia do wektora współczynników `coeff` (deklaracja, alokacja pamięci, parametry aktualne wywołania funkcji jądra i zwolnienie pamięci), za wyjątkiem kopiowania współczynników filtru z pamięci gospodarza do pamięci urządzenia. Do kopiowania do pamięci stałej służy jednak inna funkcja: `cudaMemcpyToSymbol`⁴.
5. Kompilujemy i uruchamiamy program, szacując przyspieszenie uzyskane względem wersji „naiwnej”.

³Sygnał wejściowy też tylko czytamy, ale w odróżnieniu od wektora współczynników filtru jest on duży i nie zmieści się w pamięci stałej, która nie może przekraczać 64 KiB.

⁴Sposób jej wywołania odczytamy z dokumentacji środowiska CUDA: *Start — Wszystkie Programy — NVIDIA Corporation — CUDA Documentation*. W okienku przeglądarki, które się otworzy, w lewym panelu wybieramy *CUDA API References — CUDA Runtime API — Modules — Memory Management*, po czym w prawym panelu wyszukujemy łańcuch znaków `cudaMemcpyToSymbol`.

2.1.9 Przetwarzanie sygnału techniką „kafelkową”

W dotychczasowej postaci programu każdy jego wątek obliczał iloczyn skalarny, czytając próbki sygnału wejściowego z pamięci globalnej. Na skutek tego każda próbka sygnału była czytana $n + 1$ razy, powodując duże obciążenie magistrali pamięciowej procesora graficznego. Nawet jeśli zaniedbać odczyty współczynników filtru (właśnie dlatego przeniesionych do pamięci stałej), to i tak współczynnik CGMA naszego algorytmu jest niski – w przybliżeniu równy 2 (dlaczego?). Zmniejszenie liczby wykonywanych odczytów próbek sygnału z pamięci globalnej może dać jego buforowanie w pamięci współdzielonej. Jest ona jednak mała, sygnał więc trzeba będzie buforować fragmentami („kafelkami”⁵) długości K , odpowiadającej liczbie wątków w bloku, równej liczbie obliczanych przez blok próbek sygnału wyjściowego. Zwróćmy jednak uwagę, że ze wzoru (2.2) wynika, iż do policzenia *pierwszej* z tych próbek wyjściowych potrzebna jest znajomość nie tylko bieżącej, ale także n poprzednich próbek sygnału wejściowego. „Kafelki” sygnału wejściowego muszą wobec tego mieć rozmiar $n + K$. Schemat przetwarzania sygnału z „kafelkowaniem” przedstawiono na rys. 2.1.



Rysunek 2.1: Przetwarzanie sygnału techniką „kafelkową”.

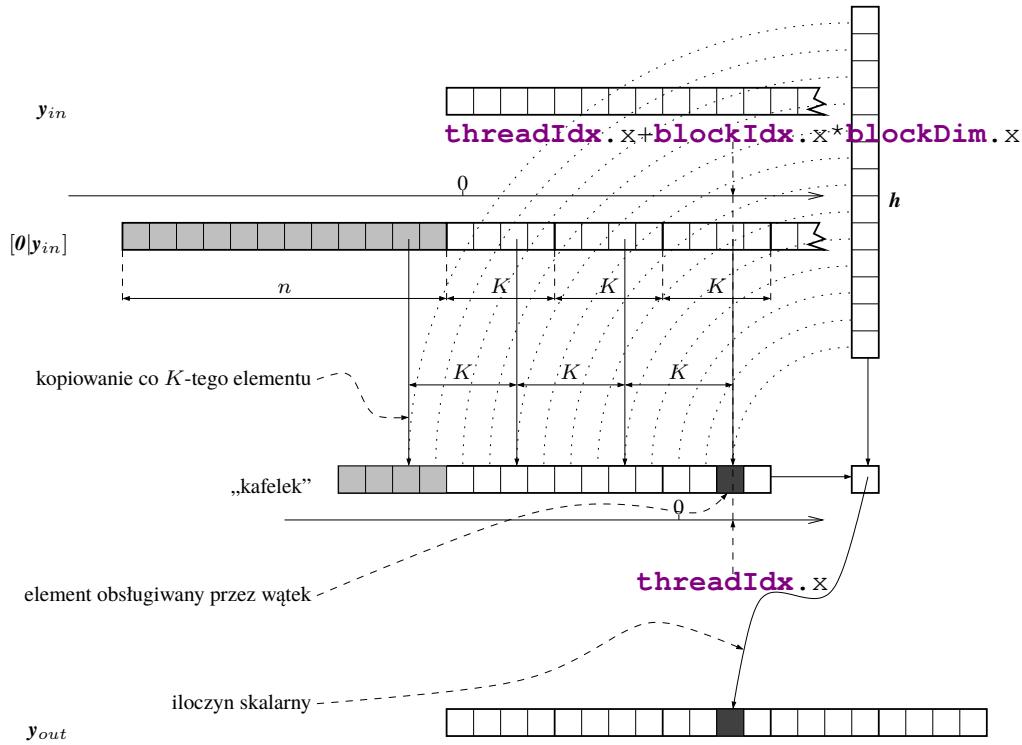
Zwróćmy uwagę, że to właśnie głównie dla potrzeb „kafelkowania” w punkcie 2.1.7 poprzedziliśmy sygnał n zerami.

„Ziarnistość” funkcji jądra musi zatem obejmować cały blok wątków, a algorytm jej działania jest dwuetapowy i składa się z kopiowania danych do pamięci współdzielonej i z wykonania obliczenia iloczynu skalarnego, przy czym oba te etapy muszą być oddzielone synchronizacją wątków bloku na barierze (por. rys. 2.2):

Require: t = identyfikator wątku w ramach bloku (**threadIdx**)

Require: b = identyfikator bloku w ramach sieci (**blockIdx**)

⁵Terminologia ta jest wprawdzie bardziej odpowiednia dla danych dwuwymiarowych, ale będziemy ją także stosować w naszym przypadku dla danych jednowymiarowych.



Rysunek 2.2: Czynności wykonywane przez pojedynczy wątek.

Require: $i = t + bK$ numer wątku w ramach całej sieci

- 1: pobierz do pamięci współdzielonej „swoją” fragment „kafelka”

$$y_{tile} = [y_{in}[bK - n], \dots, y_{in}[bK - 1], y_{in}[bK], y_{in}[bK + 1], \dots, y_{in}[bK + K - 1]]$$

wektora wejściowego, czyli co K -ty element wstecz, poczynając od bieżącego: $y_{in}[i], y_{in}[i - K], y_{in}[i - 2K], \dots$ i nie sięgając wstecz dalej niż do pierwszego elementu „kafelka” $y_{in}[bK - n]$

- 2: poczekaj, aż wszystkie wątki bloku pobiorą „swoje” elementy „kafelka” wejściowego
- 3: oblicz iloczyn skalarny odpowiedniego fragmentu „kafelka” y_{tile} przez wektor współczynników h
- 4: zapisz iloczyn skalarny do $y_{out}[i]$

Zmiany w programie obejmują zatem wyłącznie funkcję jądra.

1. Kopiujemy plik `audiofir3.cu` na nazwę `audiofir4.cu`.
2. W ciele funkcji jądra `audiofir_kernel` deklarujemy „kafelkę”

```
__shared__ float ytile[N + K];
```

i wypełniamy co K -ty jego element zgodnie z powyższym algorytmem.

3. Po pętli kopiowania danych z pamięci globalnej do pamięci współdzielonej synchronizujemy wątki bloku na barierze

```
__syncthreads();
```

aby zagwarantować, że nie zaczniemy korzystać z „nie swoich” elementów „kafelka” zanim inne wątki ich nie wypełnią.

4. W pętli liczenia iloczynu skalarnego korzystamy z odpowiednich elementów „kafelka” zamiast z wektora y_{in} z pamięci globalnej.
5. Kompilujemy i uruchamiamy program, ponownie mierząc jego przyspieszenie względem wersji „naiwnej”.

Przedstawiony algorytm znany jest w literaturze sygnałowej jako *overlap-save*⁶. *Overlap*, gdyż kolejne „kafelki” nakładają się na siebie: mają długość $n + K$ próbek, a rozmieszczone są co K próbek. *Save*, bo przy sekwencyjnej realizacji algorytmu końcówkę bieżącego „kafelka” trzeba zachować na przyszłość, gdyż wejdzie w skład następnego „kafelka”.

W zrealizowanym właśnie algorytmie, dzięki buforowaniu i wielokrotnemu wykorzystaniu raz pobranych próbek, $K(n + 1)$ mnożeń i $K(n + 1)$ dodawań przypada na $n + K$ odczytów z pamięci globalnej. Współczynnik CGMA wynosi zatem $\frac{2K(n+1)}{n+K}$ i dla $K = 512$ oraz $n = 1024$ jest równy ok. 683. Możemy zatem spodziewać się istotnego przyspieszenia względem poprzedniej wersji – sprawdźmy, czy nasze pomiary czasu potwierdzają te oczekiwania. Widać, że współczynnik CGMA będzie tym większy, im większy będzie rozmiar „kafelka”, ale ten z kolei jest ograniczony sprzętowo maksymalną liczbą wątków w bloku.

2.1.10 Statyczny rząd filtru i rozwinięcie pętli

Do tej pory rząd filtru odczytywany był z pliku i z punktu widzenia programu był zmienną, która dynamicznie zależała od danych wejściowych. Zobaczmy, jaki zysk da uczynienie rzędu filtru statycznym.

1. Kopiujemy plik `audiofir4.cu` na nazwę `audiofir5.cu`.
2. Na głównym poziomie tak utworzonego pliku zmieniamy komentarz dotyczący rzędu filtru:

```
#define N 1024 // rząd filtru FIR
```

(prawdziwe komentarze są ważne!).

3. Usuwamy rząd filtru `n` z listy parametrów formalnych funkcji jądra `audiofir_kernel` i wszystkie jego wystąpienia w ciele funkcji jądra zastępujemy stałą `N`.
4. Sprawdzany na początku funkcji `audiofir` warunek bezpieczeństwa `n <= N` zastępujemy warunkiem `n == N`.
5. Usuwamy rząd filtru `n` z listy parametrów aktualnych przy obu wywołaniach funkcji jądra `audiofir_kernel`.
6. Teraz, gdy liczba obiegów pętli obliczania iloczynu skalarnego zadana jest statycznie, można całkowicie rozwinać tę pętlę, umieszczając bezpośrednio przed instrukcją `for` linię:

⁶Por. T. Zieliński, *Od teorii do cyfrowego przetwarzania sygnałów*, Wyd. EAIiE AGH, Kraków 2002, p. 13.6.

```
#pragma unroll
```

7. Kompilujemy i uruchamiamy program, podziwiając uzyskane przyspieszenie⁷. Skąd ono się bierze? Obejrzyjmy wygenerowany przez kompilator i asembler kod maszyny fizycznej, używając programu `cuobjdump` z opcją `-sass`. Końcowy fragment otrzymanego w ten sposób kodu powinien zawierać na przemian instrukcje przesłania danej z pamięci współdzielonej (`LDS` – *Load from Shared Memory*) i instrukcje mnożenia z dodawaniem (`FFMA` – *Floating-point-32 Fused Multiply Add*)⁸. Wszystkie one mają stałe adresy i właśnie możliwość pominięcia arytmetyki adresowej dzięki statycznemu rzędowi filtru jest główną przyczyną osiągniętego przyspieszenia – doprawdy trudno lepiej napisać ten kod. Drugą przyczyną przyspieszenia jest zlikwidowanie narzutów na zapętlenie obliczeń.

2.1.11 Dodatkowe próby optymalizacji

1. Kompilujemy program z włączoną opcją `--ptxas-options=-v`. W tym celu we właściwościach projektu włączamy flagę *Configuration Properties — CUDA C/C++ — Device — Verbose PTXAS Output* i ponownie budujemy projekt. Asembler wypisze wtedy, jakich zasobów sprzętowych używa, np.:

```
ptxas : info : Used 11 registers, 3072 bytes smem, ...
```

Najistotniejsze będą dla nas dwie pierwsze pozycje: liczba wykorzystywanych przez wątek rejestrów i rozmiar wykorzystywanej przez blok pamięci współdzielonej.

2. Uruchamiamy „kalkulator wykorzystania zasobów” – jest to arkusz kalkulacyjny, który pomaga dobrać konfigurację wykonania programu zapewniającą jak najlepsze wykorzystanie zasobów (*occupancy*) wieloprocessora. W tym celu przechodzimy do lokalizacji `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\tools`. Następnie uruchamiamy znajdujący się tam arkusz kalkulacyjny `CUDA_Occupancy_Calculator.xls`. W uruchomionym arkuszu wybieramy możliwości obliczeniowe naszego procesora (dla procesorów w laboratorium jest to 7.5) i pozostawiamy domyślną wielkość pamięci współdzielonej (*Select Shared Memory Size Config (bytes)*). Następnie na podstawie wyników otrzymanych w poprzednim podpunkcie wpisujemy w odpowiednie pozycje arkusza:
 - (a) liczbę wątków na blok (*Threads Per Block*), czyli rozmiar „kafelka” K ,
 - (b) liczbę rejestrów na wątek (*Registers Per Thread*),
 - (c) rozmiar pamięci współdzielonej na blok (*Shared Memory Per Block*).

⁷Na procesorze z 16. rdzeniami CUDA autor uzyskał przyspieszenie (względem wersji CPU) $16,6\times$, a na procesorze z 448. rdzeniami CUDA – $366\times$. Oznacza to, że napisany program dobrze skaluje się względem liczby rdzeni, co jest jedną z zalet technologii CUDA.

⁸Listy rozkazów różnych wieloprocessorów strumieniowych są pokrótce opisane w dokumentacji środowiska CUDA: *Start — Wszystkie Programy — NVIDIA Corporation — CUDA Documentation*. W okienku przeglądarki, które się otworzy, w lewym panelu wybieramy *Tools — CUDA Binary Utilities — Instruction Set Reference — Maxwell Instruction Set* (dla procesora będącego na wyposażeniu laboratorium).

Patrzemy na tabelki arkusza kalkulacyjnego położone poniżej miejsc, w które wpisaliśmy parametry – czerwonym kolorem zostanie tam wskazany zasób ograniczający uzyskanie większej wydajności. Analizujemy również wykresy wykorzystania zasobów w funkcji trzech przytoczonych powyżej zasobów (liczby wątków na blok, rejestrów na wątek i bajtów pamięci współdzielonej na blok). Sprawdzamy np., czy zwiększenie rozmiaru „kafelka” do 1024 spowoduje znaczące pogorszenie wykorzystania zasobów. Uczyńmy to i sprawdźmy, czy wydajność programu rzeczywiście ulegnie zmianie. Jeśli czas nam na to pozwala, to zrobmy wykres przyspieszenia (względem wersji „naiwnej”) w funkcji K (np. od 128 do 1024 z krokiem 128).

3. Skomplikowany proces ustalania optymalnej (lub prawie optymalnej) konfiguracji wywołania funkcji jądra można sobie ułatwić, wykorzystując interfejs programistyczny (API) „kalkulatora wykorzystania zasobów”. Pozwala on na określenie rozmiaru bloku (liczby wątków) maksymalizującego wykorzystanie zasobów wieloprocessora strumieniowego oraz minimalnego rozmiaru sieci (liczby bloków) gwarantującego wykorzystanie wszystkich wieloprocessorów GPU. Maksymalizacja wykorzystania zasobów jest wprawdzie heurystycznym, ale na ogół dobrze działającym, kryterium optymalizacji konfiguracji wywołania jądra.

W celu wykorzystania tego API, w funkcji `audiofir` zaraz po wywołaniu funkcji `cudaSetDevice` wstawiamy następujący fragment kodu:

```
int minGridSize; // określenie optymalnej konfiguracji
int blockSize;
checkCudaErrors(cudaOccupancyMaxPotentialBlockSize(
    &minGridSize, &blockSize, audiofir_kernel, 0, 0));
printf("Quasi-optimal cfg.: >=%5d blocks of length %d\n",
    minGridSize, blockSize);
printf("Actual configuration: %3d blocks of length %d\n",
    (len + K - 1) / K, K);
```

Po uruchomieniu programu możemy (na podstawie powyższego wydruku) sprawdzić, czy nasza konfiguracja nie odbiega od quasi optymalnej i w razie potrzeby przededefiniować wartość zmiennej K i ew. ponownie skompilować program. Warto zwrócić uwagę, że nie musimy już tym razem odczytywać z raportu asemblera lub profilera liczby użytych rejestrów ani rozmiaru wykorzystanej pamięci współdzielonej – biblioteka CUDA odczytuje to sama z kodu binarnego funkcji jądra.

4. Przeanalizujmy jeszcze wydajność programu za pomocą profilera. Czy daje on jakieś ważne wskazówki odnośnie jakości naszego kodu i dalszych możliwości jego przyspieszenia?
5. Trzeba tu koniecznie zaznaczyć, że dalsze zwiększenie wydajności filtracji można uzyskać przez fundamentalną zmianę algorytmu. Zamiast powtarzać w pętli obliczanie iloczynu skalarnego (co w istocie prowadzi do wyznaczenia dyskretnego spłotu fragmentu sygnału ze współczynnikami filtru), można obliczyć spłot w dziedzinie częstotliwości, wykorzystując algorytm szybkiego przekształcenia Fouriera (FFT – *Fast Fourier Transform*). Takie rozwiązanie leży jednak poza zakresem niniejszego ćwiczenia.

Rozdział 3

Biblioteki wysokiego poziomu

Opis tego ćwiczenia również będzie zakładał, że Czytelnik posiadał już podczas pierwszego laboratorium podstawowe umiejętności dotyczące posługiwania się środowiskiem CUDA. W razie potrzeby zaleca się sięgnięcie do pierwszego rozdziału.

Podobnie jak dotąd, podczas laboratorium trzeba „chwalić się” prowadzącemu każdym wykonanym zadaniem i możliwie szybko zgłaszać ew. trudności w jego wykonaniu.

Przed laboratorium należy wykonać w domu zadania 3.1.1, 3.1.2, 3.2.1 i 3.2.2. Uwaga: w laboratorium jest dostępny Internet, więc kod zadania 3.1.2 można albo przynieść z domu na nośniku USB, albo zapisać na swoim koncie na jakimś, najlepiej wydziałowym, serwerze.

3.1 Biblioteki Thrust i CURAND – obliczanie wielowymiarowej całki oznaczonej metodą Monte Carlo

3.1.1 Sformułowanie problemu

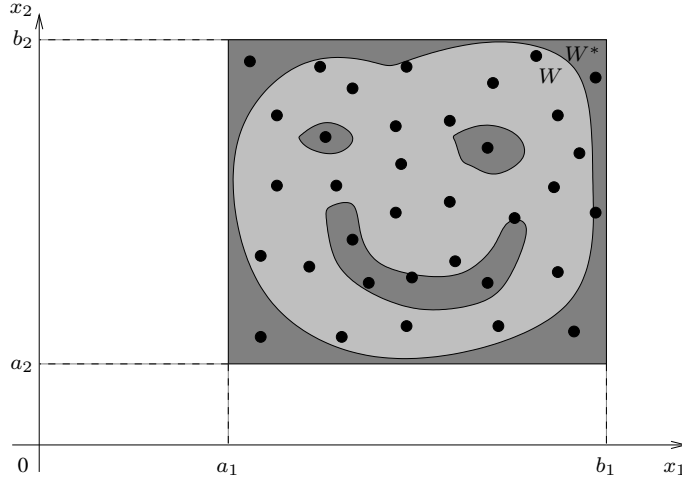
Jednym z ważniejszych zastosowań metody Monte Carlo jest numeryczne obliczanie całek oznaczonych, zwłaszcza w sytuacjach, gdy funkcja podcałkowa lub obszar całkowania są tak skomplikowane, że rozwiązanie analityczne jest niemożliwe, albo gdy całka jest wielowymiarowa. Idea całkowania metodą Monte Carlo jest opisana w literaturze obowiązkowej do niniejszego ćwiczenia:

- W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical Recipes: The Art of Scientific Computing. Third Edition in C++*, Cambridge University Press, 2007, ss.397–410.

Pozycję tę można przeczytać (we fragmencie nieprzekraczającym 30 stron za darmo) on-line pod adresem <http://apps.nrbook.com/empanel/index.html>.

Załóżmy, że chcemy scałkować funkcję $f(\mathbf{x})$, gdzie $\mathbf{x} = [x_1, \dots, x_d]^T$ (jest to więc funkcja d zmiennych), po pewnym obszarze $W \subset \mathbb{R}^d$ (por. rys. 3.1 dla przypadku dwuwymiarowego, czyli dla $d = 2$). Zamiast niej możemy scałkować funkcję

$$f_W(\mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{dla } \mathbf{x} \in W \\ 0 & \text{dla } \mathbf{x} \notin W \end{cases} \quad (3.1)$$



Rysunek 3.1: Ilustracja całkowania metodą Monte Carlo dla przypadku dwuwymiarowego.

po dowolnym (ale najlepiej jak najmniejszym) obszarze $W^* \supset W$, będącym d -wymiarowym prostopadłościem („kostką”). Zachodzi bowiem równość:

$$\int f dW = \int f_W dW^*. \quad (3.2)$$

Można w tym celu losowo wybrać N punktów $\mathbf{x}_0, \dots, \mathbf{x}_{N-1}$, gdzie $\mathbf{x}_i \in W^* \subset \mathbb{R}^d$, $i = 1, \dots, N$, równomiernie rozłożonych w obszarze W^* , a całkę z funkcji przybliżyć jej średnią wartością funkcji pomnożoną przez d -wymiarową objętość $V(W^*)$ obszaru W^* . Ponieważ obszar ten jest d -wymiarowym prostopadłościem, więc jego objętość łatwo policzyć jako iloczyn długości jego krawędzi (długości te równe są $b_k - a_k$, $k = 1, \dots, d$ – por. rys. 3.1):

$$V(W^*) = \prod_{k=1}^d (b_k - a_k). \quad (3.3)$$

Otrzymamy zatem wzór:

$$\int f dW = \int f_W dW^* \approx \langle f_W(\mathbf{x}) \rangle_{\mathbf{x} \in W^*} \cdot V(W^*), \quad (3.4)$$

gdzie wartość średnią $\langle f_W(\mathbf{x}) \rangle$ liczymy jak zwykłą średnią arytmetyczną:

$$\langle f_W(\mathbf{x}) \rangle_{\mathbf{x} \in W^*} = \frac{1}{N} \sum_{i=0}^{N-1} f_W(\mathbf{x}_i). \quad (3.5)$$

Warto zauważyć, że taka metoda obliczeniowa w naturalny i bardzo prosty sposób daje się zrównoleglić. Obliczenia w każdym punkcie są bowiem całkowicie niezależne od obliczeń w dowolnym innym punkcie. Jądro programu wykonywanego na procesorze wielordzeniowym powinno obliczać wartość funkcji podcałkowej f_W . Dla bardziej złożonych funkcji podcałkowych z łatwością będzie można uzyskać sensownie duże wartości współczynnika CGMA (*Compute to Global Memory Access ratio*), a więc implementacja metody Monte Carlo na procesorze graficznym powinna dać duże zyski czasowe.

Wybrane punkty $\mathbf{x}_0, \dots, \mathbf{x}_{N-1}$ mogą leżeć na regularnej siatce prostokątnej, ale wtedy trzeba je wszystkie wziąć pod uwagę w obliczeniach. W metodzie Monte Carlo są to punkty *pseudolosowe*, o niezależnych od siebie równomiernych rozkładach brzegowych $\mathcal{U}(a_k, b_k)$ na każdej współrzędnej x_k . Rozkłady takie łatwo jest generować i właśnie dlatego jako obszar całkowania przyjęto wielowymiarowy prostopadłościan W^* . Można też w metodzie Monte Carlo przyjąć *quasi*-losowy rozkład punktów. Nie spełniają one wtedy wprawdzie żadnych warunków statystycznych, ale na każdym etapie generacji ciągu *quasi*-losowego jego dotychczasowy podciąg stara się maksymalnie równomiernie wypełnić obszar W^* . Dzięki tej właściwości liczby *quasi*-losowe mają w tym zastosowaniu przewagę nad pseudolosowymi: w przypadku ich zastosowania niepewność przybliżenia asymptotycznie (dla dużych N) szybciej maleje wraz ze wzrostem N .

Istnieje oszacowanie niepewności standardowej przybliżenia (3.4):

$$u = V(W^*) \cdot \sqrt{\frac{\langle f_W^2 \rangle - \langle f_W \rangle^2}{N}}. \quad (3.6)$$

Wymaga ono policzenia dodatkowo wartości średniej kwadratu funkcji podcałkowej.

Przedstawione w tym punkcie całki oznaczone w przestrzeni trójwymiarowej ($d = 3$) mają szereg ważnych zastosowań w mechanice klasycznej. Przykładowe wielkości fizyczne, które za ich pomocą można obliczyć, zestawiono w tab. 3.1.

Tabela 3.1: Definicje przykładowych wielkości mechanicznych (symbol $\rho(x, y, z)$ oznacza lokalną gęstość masy w punkcie (x, y, z))

Wielkość fizyczna	Definicja: $f(x, y, z) =$
objętość	1
masa	$\rho(x, y, z)$
współrzędna x środka masy \times masa	$x\rho(x, y, z)$
współrzędna y środka masy \times masa	$y\rho(x, y, z)$
współrzędna z środka masy \times masa	$z\rho(x, y, z)$
moment bezwładności względem osi Ox	$(y^2 + z^2)\rho(x, y, z)$
moment bezwładności względem osi Oy	$(x^2 + z^2)\rho(x, y, z)$
moment bezwładności względem osi Oz	$(x^2 + y^2)\rho(x, y, z)$

3.1.2 Prototypowy program na procesor CPU – obliczanie objętości kuli

Tym razem nie będziemy świadomie pisali „kiepskiego” programu, aby go potem ulepszać. Od początku przy tworzeniu kodu źródłowego uwzględnimy podstawowe techniki optymalizacji charakterystyczne dla biblioteki Thrust implementowanej na procesorach graficznych:

- Preferowanie struktury tablic zamiast tablicy struktur. Nie będziemy zatem używali wektora elementów typu `float3`, tylko trzy niezależne wektory elementów typu `float` (nasz program jest na tyle prosty, że nie trzeba tych trzech wektorów łączyć w jedną strukturę).

- Wykorzystanie nietypowych iteratorów. Zastosujemy np. `zip_iterator` do wirtualnego grupowania odpowiadających sobie elementów tych trzech wektorów w jeden punkt przestrzeni trójwymiarowej.
- Łączenie prostych operacji w bardziej złożone. Przykładowo, zamiast oddzielnie wykonywać obliczanie wartości funkcji podcałkowej (transformacja punktów \mathbb{R}^3 funktorem realizującym funkcję f_W) i całkowanie (redukcja z domyślnym funktorem sumującym), zastosujemy łączący je algorytm `transform_reduce` biblioteki Thrust.

A zatem przystąpmy do pracy.

1. Tworzymy plik `mcint.cu` (nazwa pochodzi od *Monte Carlo INTEGRation*) o następującej zawartości:



```
# define THRUST_DEVICE_SYSTEM THRUST_DEVICE_SYSTEM_OMP

#include <thrust/functional.h> // function objects & tools
#include <thrust/random.h>
#include <thrust/random/uniform_real_distribution.h>
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/iterator/zip_iterator.h>

/*****/

#ifdef _WIN32

#define WINDOWS_LEAN_AND_MEAN
#include <windows.h>

typedef LARGE_INTEGER app_timer_t;

static inline void timer(app_timer_t *t_ptr)
{
#ifdef __CUDACC__
    checkCudaErrors(cudaDeviceSynchronize());
#endif
    QueryPerformanceCounter(t_ptr);
}

double elapsed_time(app_timer_t start, app_timer_t stop)
{
    LARGE_INTEGER clk_freq;
    QueryPerformanceFrequency(&clk_freq);
    return (stop.QuadPart - start.QuadPart) /
        (double) clk_freq.QuadPart * 1e3;
}

#else
```

```

#include <time.h> /* requires linking with rt library
                  (command line option -lrt) */

typedef struct timespec app_timer_t;

static inline void timer(app_timer_t *t_ptr)
{
#ifdef __CUDACC__
    checkCudaErrors(cudaDeviceSynchronize());
#endif
    clock_gettime(CLOCK_MONOTONIC, t_ptr);
}

double elapsed_time(app_timer_t start, app_timer_t stop)
{
    return 1e+3 * (stop.tv_sec - start.tv_sec) +
           1e-6 * (stop.tv_nsec - start.tv_nsec);
}

#endif

/*****

class randuni :
    public thrust::unary_function<unsigned long long, float>
{
private:
    thrust::default_random_engine rng;
    thrust::uniform_real_distribution<float> uni;
public:
    randuni(unsigned int seed, float a=0.0f, float b=1.0f) :
        rng(seed), uni(a, b) {}
    __host__ __device__
    float operator()(unsigned long long i)
    {
        rng.discard(i); // odrzuć liczby z "poprzednich" wątków
        return uni(rng);
    }
};

*****/

typedef thrust::tuple<float, float, float> point3D;

struct fun : public thrust::unary_function<point3D, float>
{
    __host__ __device__
    float operator()(const point3D &p) const

```

```

{
    float x = thrust::get<0>(p);
    float y = thrust::get<1>(p);
    float z = thrust::get<2>(p);
    return // TODO: Obliczenie wartości f-cji ch-nej kuli
}
};

/*****

int main()
{
    app_timer_t t0, t1, t2, t3;
    float integral;
    timer(&t0); //-----
    thrust::device_vector<float> x(1000),
        y(x.size()), z(x.size());
    timer(&t1); //-----
    // TODO: Generacja niezależnych wektorów losowych x,y,z
    timer(&t2); //-----
    // TODO: Całkowanie numeryczne metodą Monte-Carlo
    integral = ...
    timer(&t3); //-----

    using std::cout;
    using std::endl;
    cout<<"pi = "<<0.75f * integral<<std::endl; // =pi?
    cout<<"Inicjacja:  "<<elapsed_time(t0, t1)<<" ms"<<endl;
    cout<<"Generacja:  "<<elapsed_time(t1, t2)<<" ms"<<endl;
    cout<<"Integracja:  "<<elapsed_time(t2, t3)<<" ms"<<endl;
    cout<<"R A Z E M :  "<<elapsed_time(t0, t3)<<" ms"<<endl;
#ifdef _WIN32
    if (IsDebuggerPresent()) getchar();
#endif
    return 0;
}

```

Plik ten nie jest jeszcze kompletnym źródłem, lecz jedynie szkieletem docelowego programu, zaraz go uzupełnimy w miejscach wyróżnionych komentarzem *// TODO:*.

Na początku tego pliku następuje wskazanie interfejsu OpenMP (*Open Multi-Processing*) jako „silnika” naszej aplikacji Thrust. Jak omawialiśmy na wykładzie, biblioteka Thrust, której użyjemy do implementacji całkowania metodą Monte Carlo, nie wymaga procesora karty graficznej i może być wykonywana na procesorze CPU z biblioteką OpenMP. Świetnie się to nadaje do napisania i uruchomienia prototypu programu.

Z kolei następuje sekcja załączająca niezbędne pliki nagłówkowe biblioteki Thrust. Jeśli użytkownik w trakcie rozwoju programu szkielekowego skorzysta z jakiejś nowej funkcjonalności tej biblioteki, to trzeba będzie odpowiednio uzupełnić tę sekcję.

Dalej występują definicje pomocniczych funkcji do pomiaru czasu wykonania progra-

mu. Są one bardzo podobne do tych z poprzedniego ćwiczenia, ale nie zawierają tym razem „wbudowanych” wydruków i obliczania wydajności, gdyż będą wykorzystywane w różnych kontekstach (do pomiaru czasu wykonania różnych fragmentów programu). Zawierają za to synchronizację gospodarza i urządzenia, aby umożliwić pomiar czasu także dla operacji asynchronicznie uruchamianych przez gospodarza na urządzeniu.

Kolejna sekcja kodu zawiera definicję klasy `randuni`. Obiekty tej klasy są funktorami, których stan obejmuje „silnik” generatora pseudolosowego `rng` oraz sam generator `uni` liczb pseudolosowych o rozkładzie równomiernym (*uniform*) na przedziale (a, b) , czyli rozkładu $\mathcal{U}(a, b)$. Klasa ta jest niemal identyczna jak dokładnie omawiana na wykładzie klasa `randn`, generująca liczby o rozkładzie normalnym.

Następna część programu to już nasza funkcja podcałkowa f_W , nazwana tu `fun`. Na początku definiujemy typ `point3D` jako krotkę trzech rzeczywistych współrzędnych punktu w przestrzeni trójwymiarowej. Potem definiujemy funkcję podcałkową `fun` jako funktor jednoargumentowy `point3D` → `float`, czyli $\mathbb{R}^3 \rightarrow \mathbb{R}$, który każdemu punktowi \mathbf{x} przestrzeni przyporządkowuje wartość $f_W(\mathbf{x})$. Przeciążamy zatem operator `()`, a w ciele jego definicji za pomocą funkcji `thrust::get` „wyłuskujemy” kolejne elementy krotki wejściowej, czyli współrzędne x, y i z .

W niniejszym punkcie będziemy chcieli obliczyć objętość kuli $S(\mathbf{0}, 1)$ o środku w początku układu współrzędnych ($\mathbf{0}$) i o jednostkowym promieniu ($r = 1$). Zgodnie z uwagami z p. 3.1.1, f_W jest po prostu funkcją charakterystyczną kuli S :

$$f_W(x, y, z) = \begin{cases} 1 & \text{gdy } x^2 + y^2 + z^2 \leq 1 \\ 0 & \text{gdy } x^2 + y^2 + z^2 > 1 \end{cases} \quad (3.7)$$

Odpowiedni kod należy dopisać w instrukcji `return`. Zauważmy, że dla wygody współrzędne x_1, x_2, x_3 wektora \mathbf{x} oznaczyliśmy tu odpowiednio przez x, y, z .

Wreszcie ostatnia część kodu to program główny. Zadeklarowane są w nim cztery punkty pomiaru czasu:

- przed pierwszą operacją biblioteki Thrust,
- po inicjacji wektorów współrzędnych punktów,
- po generacji liczb pseudolosowych,
- po zakończeniu pracy biblioteki Thrust, czyli po całkowaniu

oraz zmienna do przechowania wartości całki. Potem następuje deklaracja wektora \mathbf{x} mającego 1000 elementów oraz wektorów \mathbf{y} i \mathbf{z} takiego samego rozmiaru, a po komentarzach opisujących, co należy zrobić – wydruk wyników. Zauważmy, że nie drukujemy wprost wartości obliczonej całki, tylko mnożymy ją przez $\frac{3}{4}$. Dlaczego? Objętość kuli wynosi $\frac{4}{3}\pi r^3$, więc dla $r = 1$ wyprowadzimy w ten sposób łatwo rozpoznawalną liczbę π .

Uzupełniamy brakujące fragmenty kodu źródłowego funkcji `main`. W pierwszym z nich dokonamy generacji trzech *niezależnych* wektorów losowych \mathbf{x}, \mathbf{y} , i \mathbf{z} . W tym celu powołujemy do życia trzy *różne* obiekty generatorów typu `randuni`, z różnymi wartościami załączków ciągu pseudolosowego. Z dużym prawdopodobieństwem zapewni to uzyskanie trzech niezależnych ciągów pseudolosowych. Każdy z generatorów powinien

generować dane o rozkładzie $\mathcal{U}(-1, 1)$, gdyż to właśnie sześcian o boku 2 i o środku w początku układu współrzędnych jest najlepszym dla naszej kuli obszarem całkowania W^* .

Pamiętajmy, że w równoległej realizacji naszego programu teoretycznie (na sprzęcie mającym nieskończenie wiele rdzeni) każdy wątek (generujący jedną liczbę pseudolosową) wykona się na innym rdzeniu – innymi słowy – każdy rdzeń wygeneruje tylko jedną liczbę pseudolosową (*tą samą dla wszystkich rdzeni*; będzie to pierwszy element ciągu pseudolosowego). Aby tego efektu uniknąć, funktor `randuni` ma parametr, mówiący, o który z kolei element ciągu pseudolosowego nam chodzi. Dlatego generacja jest w istocie *transformacją* ciągu liczb $0, \dots, N - 1$ w elementy ciągu pseudolosowego o odpowiednich indeksach. Należy ją zatem zakodować z użyciem algorytmu `transform` biblioteki Thrust, wykorzystując jako argument tej transformacji kolejny nietypowy iterator, a mianowicie `counting_iterator`.

Ostatnim brakującym fragmentem kodu źródłowego będzie właściwe całkowanie. Jak już wspominaliśmy, dokonamy tej operacji łącząc w jedno *transformację* wejściowego zbioru punktów w przestrzeni trójwymiarowej funktorem `fun` z *redukcją* wyników wykorzystującą (domyślny) funktor `plus`. Wynik tych działań jest *sumą*, a nie średnią wartością funkcji podcałkowych, musimy go więc jeszcze podzielić przez liczbę punktów, a następnie, zgodnie ze wzorem (3.4), pomnożyć przez objętość sześcianu W^* , równą 8. Aby móc dostawać się „równocześnie” do trzech współrzędnych punktu w przestrzeni, zapisanych wszak w niezależnych i niepowiązanych ze sobą wektorach x , y i z , iteratory tych wektorów musimy połączyć ze sobą w trójelementowe krotki za pomocą funkcji `make_zip_iterator` i `make_tuple` biblioteki Thrust.

- Wywołujemy zintegrowane środowisko programistyczne Microsoft Visual Studio 2022 i tworzymy w nim nowy, pusty projekt C++ o nazwie *mcint*. Pamiętajmy, aby był to projekt 64-bitowy. Dodajemy do projektu jako plik źródłowy (*Source Files*) przed chwilą utworzony `mcint.cu`. W ogólnych właściwościach pliku, w zakładce *General*, nadajemy polu *Item Type* wartość *C/C++ compiler*. Jest to konieczne, gdyż projekt utworzyliśmy jako projekt C++, a nie projekt CUDA i kompilator nie wiedziałby, jak potraktować plik o rozszerzeniu `.cu`.
- We właściwościach projektu ustawiamy *Configuration Properties — C/C++ — Language — OpenMP Support* na *Yes*. Nasza prototypowa wersja programu będzie bowiem korzystała z systemu OpenMP, wsparcie którego trzeba w kompilatorze właśnie w ten sposób włączyć. Naszym *urządzeniem* będzie zatem zbiór rdzeni procesora CPU – na każdym rdzeniu zostanie przez system OpenMP uruchomiony jeden wątek.
- Ponownie we właściwościach projektu edytujemy pozycję *Configuration Properties — VC++ Directories — Include Directories* dopisując do niej ścieżkę do biblioteki Thrust. Jeśli na komputerze jest już zainstalowane środowisko CUDA, będzie to ścieżka `$(CUDA_PATH)/include`. W przeciwnym przypadku trzeba uprzednio ze strony <http://thrust.github.com/> ściągnąć najnowszą wersję biblioteki Thrust, zainstalować ją gdzieś na komputerze i wskazać miejsce tej instalacji.
- Kompilujemy program i uruchamiamy go, aż do uzyskania w wyniku jego pracy liczby zbliżonej do π .

3.1.3 Przeniesienie programu na procesor GPU

Obecnie dokonamy przeniesienia uruchomionego w domu oprogramowania na procesor graficzny. W tym celu wykonujemy następujące kroki:

1. Aby nie mieć problemu z ustawieniami ścieżek i innych właściwości projektu środowiska Microsoft Visual Studio, najwygodniej będzie utworzyć *nowy* projekt. Klikając prawym klawiszem myszy na nazwę naszego rozwiązania (*Solution*) w polu *Solution Explorer* wybieramy *Add — New Project*, a w okienku, które się pojawi, w zakładce *Installed* rozwijamy *NVIDIA* i wybieramy pozycję *CUDA (CUDA 12.6 Runtime)*. Następnie wybieramy nazwę projektu i jego lokalizację. Pamiętajmy, aby jako platformę rozwiązania wybrać projekt 64-bitowy. Otwiera się projekt z jednym szablonowym plikiem `kernel.cu`. Klikając prawym klawiszem myszy nazwę tego pliku w polu *Solution Explorer* wybieramy *Remove*, a potem *Delete*. Wreszcie klikając prawym klawiszem myszy nazwę naszego nowego projektu wybieramy *Add — Existing Item* i wskazujemy jako plik do dodania do projektu przygotowany w ramach poprzedniego punktu plik `mcint.cu`.
2. Na początku pliku źródłowego `mcint.cu` usuwamy (ewentualnie uwarunkowujemy *niezdefiniowaniem* symbolu `__CUDACC__`) definicję `THRUST_DEVICE_SYSTEM`. W ten sposób wracamy do domyślnego „silnika” biblioteki Thrust, czyli do środowiska CUDA.
3. Na początku pliku źródłowego `mcint.cu` dodajemy standardowe linie:

```
#include <cuda_runtime.h>
#include <helper_cuda.h>
```

4. We właściwościach projektu do opcji *Configuration Properties — VC++ Directories — Include Directories* dodajemy pozycję `$(NVCUDASAMPLES_ROOT)/common/inc` dla wszystkich konfiguracji projektu. Jest ona niezbędna do znalezienia pliku `helper_cuda.h`.
5. Kompilujemy program w konfiguracji uruchomieniowej (*Debug*) i uruchamiamy go, upewniając się, że w wyniku jego pracy uzyskamy ponownie liczbę zbliżoną do π .
6. Kompilujemy program w konfiguracji produkcyjnej (*Release*) i uruchamiamy go, upewniając się, że w wyniku jego pracy uzyskamy znowu liczbę zbliżoną do π , ale także notując czasy trwania poszczególnych etapów programu. Zwracamy uwagę na relatywnie duży czas generacji liczb pseudolosowych.

3.1.4 Generacja liczb pseudolosowych za pomocą biblioteki CURAND

Aby zmniejszyć duży czas generacji liczb pseudolosowych, spowodowany głównie linią:

```
rng.discard(i); // odrzuć liczby z "poprzednich" wątków
```

w kodzie funktora `randuni`, będziemy musieli skorzystać z biblioteki CURAND firmy NVIDIA, dostarczanej wraz ze środowiskiem CUDA. Aby to uczynić, zrobimy następujące zmiany w pliku źródłowym `mcint.cu`:

1. Na początku pliku dodajemy linię:

```
#include <curand.h>
```

dołączając w ten sposób do niego wszelkie definicje biblioteki CURAND.

2. Usuwamy z pliku¹ definicję klasy `randuni`. Zamiast niej wpisujemy do programu definicję innego funktora, który przekształci nam liczby pseudolosowe o rozkładzie $\mathcal{U}(0, 1)$ (a tylko takie potrafi generować biblioteka CURAND) na liczby o rozkładzie $\mathcal{U}(-1, 1)$. Szkic definicji tego funktora wygląda następująco:

```
class uniformAB :
public thrust::unary_function<float, float>
{
private:
    float a, b;
public:
    uniformAB(float _a, float _b) :
        a(_a), b(_b) {}
    __host__ __device__ float operator()(float x)
    {
        return // TODO: Przekształcenie liczby losowej x
               // z przedziału (0,1) na liczbę losową
               // z przedziału (a,b)
    }
};
```

3. Usuwamy całą sekcję pliku odpowiedzialną za generację liczb pseudolosowych siłami biblioteki Thrust (pomiędzy odczytami czasu do stoperów `t1` i `t2`). Zamiast tego wpisujemy następujące fragmenty kodu:
 - (a) Deklarację trzech zmiennych typu `curandGenerator_t`, które będą trzema niezależnymi „silnikami” generatorów pseudolosowych dla współrzędnych x , y i z .
 - (b) Utworzenie w tych zmiennych trzech „silników” generatorów za pomocą funkcji `curandCreateGenerator()`. Skorzystamy z domyślnego „silnika”, oznaczonego symbolem `CURAND_RNG_PSEUDO_DEFAULT`.
 - (c) Ustawienie na różne wartości załączków ciągów pseudolosowych dla każdego z tych trzech „silników”. Wykorzystamy w tym celu funkcję biblioteki CURAND o nazwie `curandSetPseudoRandomGeneratorSeed()`.
 - (d) Generacja danych pseudolosowych dla współrzędnych x , y i z . Ponieważ używana w tym celu funkcja `curandGenerateUniform()` biblioteki CURAND przyjmuje jako jeden ze swych parametrów zwyczajny wskaźnik do pamięci urządzenia, więc musimy zrzutować adres początku (zerowego elementu) każdego z wektorów x , y i z (zdefiniowanych wszak jako wektory biblioteki Thrust) za pomocą funkcji `raw_pointer_cast()` biblioteki Thrust.
 - (e) Zwolnienie zasobów generatorów pseudolosowych za pomocą funkcji biblioteki CURAND o nazwie `curandDestroyGenerator()`.

¹Warto na wszelki wypadek zachować dotychczasową wersję tego pliku.

- (f) Zadeklarowanie funktora klasy `uniformAB`, który przekształci liczby pseudolosowe o rozkładzie $\mathcal{U}(0, 1)$ na liczby o rozkładzie $\mathcal{U}(-1, 1)$.
- (g) Wywołanie algorytmu `transform` biblioteki Thrust oddzielnie dla każdego z wektorów `x`, `y` i `z`, który za pomocą powołanego do życia w poprzednim punkcie funktora przekształci „w miejscu”² każdy z tych wektorów na rozkład równomierny na przedziale $(-1, 1)$.

Dokładniejsze informacje odnośnie wszystkich wymienionych powyżej funkcji można znaleźć w dokumentacji biblioteki CURAND: *Start — Wszystkie Programy — NVIDIA Corporation — CUDA Documentation*. W okienku przeglądarki, które się otworzy, w lewym panelu wybieramy *CURAND*, po czym w prawym panelu wyszukujemy nazwę interesującej nas funkcji.

Pozostała część programu, w tym samo całkowanie za pomocą transformacji i redukcji biblioteki Thrust, nie ulega żadnej zmianie.

4. We właściwościach projektu do opcji *Configuration Properties — Linker — Input — Additional Dependencies* dodajemy pozycję `curand.lib` dla wszystkich konfiguracji projektu, aby prawidłowo skonsolidować nasz projekt z biblioteką CURAND.
5. Kompilujemy program w konfiguracji uruchomieniowej (*Debug*) i uruchamiamy go, upewniając się, że w wyniku jego pracy uzyskamy ponownie liczbę zbliżoną do π .
6. Kompilujemy program w konfiguracji produkcyjnej (*Release*) i uruchamiamy go, upewniając się, że w wyniku jego pracy uzyskamy znowu liczbę zbliżoną do π , ale także notując czasy trwania poszczególnych etapów programu. Zwracamy uwagę na wyraźne skrócenie czasu generacji liczb pseudolosowych w stosunku do biblioteki Thrust dla dostatecznie dużych rozmiarów generowanych wektorów.

3.1.5 Zastąpienie liczb pseudolosowych quasi-losowymi

Jak już pisaliśmy w p. 3.1.1, w metodzie Monte Carlo często korzystnie jest wykorzystać liczby quasi-losowe zamiast pseudolosowych.

1. Zapamiętujemy gdzieś na wszelki wypadek dotychczasową wersję programu.
2. W generatorze pseudolosowym uzyskaliśmy trzy niezależne strumienie liczb posługując się trzema generatorami, każdy z innym załączkiem ciągu pseudolosowego. Generatory quasi-losowe nie mają załączka i zawsze startują od tej samej wartości. Generatorowi takiemu podaje się wymiarowość d przestrzeni, po czym generuje on $d \cdot N$ liczb quasi-losowych: pierwsze N z nich odpowiada pierwszej współrzędnej N punktów, kolejne N liczb – drugiej współrzędnej, itd. Musimy mieć zatem jeden dodatkowy wektor o 3-krotnie większej długości:

```
thrust::device_vector<float> xyz(3 * x.size());
```

²Przekształcenie „w miejscu” oznacza, że dany wektor jest zarówno źródłem, jak i odbiornikiem danych dla algorytmu transformacji.

oraz powołać do życia jeden generator „losowy” zamiast trzech. Ostatnim parametrem funkcji `curandCreateGenerator()` powinno być `CURAND_RNG_QUASI_DEFAULT` zamiast `CURAND_RNG_PSEUDO_DEFAULT`.

3. Usuwamy z programu wywołania funkcji `curandSetPseudoRandomGeneratorSeed()`. Zamiast tego za pomocą funkcji `curandSetQuasiRandomGeneratorDimensions` ustawiamy wymiarowość przestrzeni na 3.
4. Generujemy $3 * x.size()$ liczb quasi-losowych. Wykorzystujemy w tym celu, tak jak poprzednio, funkcję `curandGenerateUniform`, po czym usuwamy nasz generator (`curandDestroyGenerator`).
5. Wykorzystując algorytm `copy` albo `copy_n` biblioteki Thrust³ kopiujemy pierwszą $\frac{1}{3}$ wektora `xyz` do wektora `x`, kolejną $\frac{1}{3}$ do wektora `y`, a ostatnią $\frac{1}{3}$ – do wektora `z`. Dalej następuje transformacja danych na rozkład $\mathcal{U}(-1, 1)$, realizowana identycznie jak poprzednio.
6. Kompilujemy program w konfiguracji produkcyjnej (*Release*) i uruchamiamy go, upewniając się, że w wyniku jego pracy uzyskamy jako średnią wartość obliczonej całki liczbę zbliżoną do π .
7. Powtarzamy całkowanie dla $N = 10^n$, gdzie $n \in \{3, 4, 5, 6, 7\}$. Rysujemy wykres $\Delta(n)$ różnicy między maksymalną i minimalną wartością obliczonej całki w funkcji n i próbujemy na tej podstawie oszacować wykładnik α w zależności empirycznej $\Delta \sim n^{-\alpha}$. Można oczywiście w tym celu ponownie wykorzystać program MATLAB.

Czy generator liczb quasi-losowych z biblioteki CURAND spełnił pokładane w nim nadzieje na uzyskanie asymptotycznie szybszego wzrostu dokładności obliczeń ze wzrostem liczby punktów N ?

3.1.6 Liczenie objętości innych brył

Od początku aż do tej pory liczyliśmy objętość kuli. Tym razem sięgniemy do innych brył.

1. Na podstawie doświadczeń z poprzednich dwóch punktów tak dobieramy parametry algorytmu Monte Carlo, aby uzyskać jak największą dokładność obliczeń całki przy zachowaniu rozsądnego czasu obliczeń.
2. Rezygnujemy z wielokrotnego powtarzania obliczenia całki i zbierania statystyk.
3. Modyfikujemy odpowiednio funkcję całkowaną f_W , tak aby odpowiadała ona jednemu spośród poniższych obiektów, wybranemu przez prowadzącego laboratorium⁴:

- (a) Fragment torusa przedstawiony (wraz ze wzorami) na s. 400 książki *Numerical Recipes* (<http://apps.nrbook.com/empanel/index.html>).

³Opis tych algorytmów można znaleźć na stronie http://thrust.github.com/doc/group_copying.html.

⁴Oczywiście nieograniczona fantazja prowadzącego może mu podpowiedzieć także i inne, jeszcze ciekawsze, bryły.

- (b) Kula o promieniu $r = \pi$ z wydrążonym centralnie cylindrycznym otworem o promieniu $r_{\text{otw}} = \frac{\sqrt{2}}{2}$.
 - (c) Grubościenna rura cylindryczna o długości $h = 5$, promieniu wewnętrznym $r_1 = 2$ i zewnętrznym $r_2 = 3$.
 - (d) Stożek o promieniu podstawy $r = 1$ i wysokości $h = 4$.
 - (e) „Saturn z pierścieniami”, czyli kula o promieniu $r_1 = 2$ złączona razem z walcem o promieniu podstawy $r_2 = 3$ i o wysokości $h = 1$ w ten sposób, że środki masy obu obiektów pokrywają się.
4. Przynależność do tych bardziej złożonych obiektów często określa się koniunkcją lub alternatywą kilku warunków. Warto wtedy sprawdzić wpływ na wydajność programu tricku programistycznego wykorzystującego wewnętrzną reprezentację typu `bool`: wartości logicznej `false` odpowiada liczba 0, a wartości logicznej `true` – liczba 1. Zamiast występującej w programie w ramach instrukcji warunkowej `if` albo operatora trójargumentowego `?:` operacji koniunkcji logicznej można niekiedy zastosować mnożenie arytmetyczne odpowiednich predykatów.

3.2 Biblioteka CUBLAS – odtwarzanie informacji ze szczątkowych danych uzyskanych metodą kompresywnego próbkowania

W niniejszym punkcie nie będziemy zasadniczo dokonywali optymalizacji czasowej programu na procesorze graficznym. Aby zwiększyć czytelność programów zostały z nich nawet usunięte instrukcje do pomiaru czasu wykonania. Naszym celem będzie jedynie nauczenie się „tłumaczenia” programu z zakresu algebry liniowej, zapisanego w języku macierzowym wysokiego poziomu (MATLAB), na serię wywołań funkcji biblioteki CUBLAS.

3.2.1 Sformułowanie problemu

W ostatnich latach ogromną popularność zyskują metody odtwarzania informacji z zupełnie szczątkowych, wydawałoby się, fragmentów. Metody te noszą angielską nazwę *compressive sensing* albo *compressed sampling*. W jakich sytuacjach metody te działają? Gdy przedstawienie sygnału w pewnej bazie jest „rzadkie”, tzn. gdy w tej bazie da się go reprezentować z kilkoma tylko współczynnikami istotnie różnymi od zera.

Idea takiego „oszczędnego próbkowania” jest dość przystępnie opisana w artykule:

- S. Engelberg: *Compressive Sensing [Instrumentation Notes]*, IEEE Instrumentation & Measurement Magazine, vol. 15, no. 1, Feb. 2012, ss.42–46.

Pozycję tę można przeczytać w wersji elektronicznej za pośrednictwem serwera biblioteki głównej Politechniki Warszawskiej.

Przykładem takiego sygnału jest utwór muzyczny wykonywany na fortepianie solo. Jest on „rzadki” w dziedzinie częstotliwości. Po pierwsze – z całego continuum częstotliwości fortepian może wydobyć tylko 88 dyskretnych częstotliwości odpowiadających poszczególnym klawiszom oraz ich harmoniczne. Po drugie – w każdej chwili pianista przyciska typowo

co najwyżej kilka klawiszy na raz, wybierając z tego niewielkiego zestawu dopuszczalnych częstotliwości małą liczbę dźwięków. Pozwala to na odtworzenie (przybliżone, oczywiście) dźwięku fortepianu na podstawie raptem kilkuset próbek na sekundę. Jest to o ponad rząd wielkości mniej niż wynikałoby z fundamentalnego twierdzenia Shannona o próbkowaniu.

Na laboratorium rozważymy inny przykład – sygnału „rzadkiego” w dziedzinie czasu. Będzie to sygnał dyskretny w czasie, opisujący dobowy opad deszczu x wyrażony w [mm] w funkcji numeru dnia n . W naszym klimacie zdecydowana większość dni w roku jest bez opadów i wartość sygnału x_n będzie w te dni zerowa. Oznaczmy przez N długość sygnału (np. dla rocznych obserwacji $N = 365$), a przez K liczbę jego niezerowych próbek. Dla $K \ll N$ będzie można mówić o sygnale „rzadkim”.

Wyobraźmy sobie, że chcemy zmierzyć codzienne wielkość opadów w zupełnie odludnym miejscu. Nikt tam nie mieszka i nie ma komu obserwować pogody. Jak zapamiętać fakt, że padał deszcz i ile go spadło? To proste – postawić odkryty kubełek (o pionowych ściankach) i codziennie o ustalonej porze mierzyć wysokość poziomu wody y_n w tym kubelku. Różnica dwóch kolejnych odczytów $y_n - y_{n-1}$ będzie wartością sygnału dobowego opadu w danym dniu x_n .

Ale komu by się chciało jeździć codziennie na takie odludzie? W najlepszym wypadku obsługa naszej stacji meteorologicznej wpadnie tam raz na kilka dni, i to w nieregularnych, całkowicie losowych terminach. Nie można liczyć na to, że ktoś tam będzie częściej niż średnio raz w tygodniu. Liczba obserwacji M będzie zatem istotnie mniejsza niż liczba interesujących nas próbek sygnału N . Czy z tak szczątkowych danych w ogóle da się odtworzyć kompletny sygnał x_n ? I to biorąc pod uwagę fakt, że przecież woda z kubelka paruje? Okazuje się, że w wielu przypadkach i z dość dobrym przybliżeniem odpowiedzi na oba te pytania są pozytywne!

Nasz system pomiarów, na początek przy założeniu *codziennych* odczytów, możemy modelować odpowiednio dobraną „pełną” macierzą A_p rozmiaru $N \times N$. Gdyby nie było parowania wody, byłaby to macierz dolna trójkątna o wszystkich elementach równych jedności. Wówczas bowiem poziom wody w kubelku y_n byłby sumą wszystkich opadów dziennych x_n od zarania dziejów (czyli od postawienia kubelka):

$$y_n = \sum_{i=1}^n x_i. \quad (3.8)$$

Zwróćmy uwagę, że jest to operacja sumy prefiksowej (skanowania) znana nam już jako algorytm biblioteki Thrust. W języku macierzy można ją zapisać następująco:

$$\mathbf{y}_p = \mathbf{A}_p \mathbf{x}, \quad (3.9)$$

gdzie $\mathbf{x} = [x_1, \dots, x_N]^T$ oraz

$$\mathbf{A}_p = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & 0 & \dots & 0 \\ 1 & 1 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & 1 & 1 & 1 & \dots & 1 \end{bmatrix}. \quad (3.10)$$

Wektor $\mathbf{y}_p = [y_1, \dots, y_N]^T$ jest „pełnym” wektorem obserwacji.

Jeśli dopuścimy parowanie wody, przyjmując dla uproszczenia wykładniczy model zaniżania wody w kubelku ze stałą czasową τ równą 30 dni, to macierz A_p nadal będzie dolną trójkątną macierzą Toeplitza (czyli o jednakowych elementach wzdłuż każdej przekątnej), ale wartości jej elementów (poza główną przekątną) nie będą już jedynkami. Wszystkie elementy k -tej przekątnej poniżej diagonalą będą miały wartość $e^{-k/\tau}$.

Jak teraz dołączyć do tego modelu rzadkość i nieregularność wizyt osoby odczytującej poziom wody w kubelku? Otóż wystarczy wybrać losowy zestaw M wierszy macierzy A_p , tworząc w ten sposób macierz A o rozmiarach $M \times N$. Będziemy ją nazywać macierzą pomiaru, a wektor

$$\mathbf{y} = \mathbf{A}\mathbf{x} \quad (3.11)$$

określimy mianem wektora obserwacji. Wykorzystując „rzadkość” sygnału x_n , można będzie go odtworzyć z tak niekompletnego (znacznie krótszego od \mathbf{y}_p) wektora \mathbf{y} .

Dla potrzeb obecnego ćwiczenia wykorzystamy w tym celu bardzo prosty (i przez to nie najlepszy, ale za to dający się łatwo zaprogramować) heurystyczny algorytm kroczącego dopasowania (*Matching Pursuit*). Jest to algorytm zachłanny, który w każdym iteracyjnie powtarzanym kroku stara się tak dobrać datę kolejnego deszczowego dnia i wielkość opadu w tym dniu, aby jak najbardziej zmniejszyć normę euklidesową resztkowego sygnału błędu \mathbf{r} . Iteracje kończą się, gdy przekroczymy ich limit, albo gdy sygnał błędu będzie pomijalnie mały.

A oto algorytm wyznaczania N -elementowego sygnału zrekonstruowanego $\hat{\mathbf{x}}$:

Require: $\mathbf{A}, \mathbf{y}, t_{\max}, \varepsilon$

```

1:  $t \leftarrow 1, \hat{\mathbf{x}} \leftarrow \mathbf{0}, \mathbf{r} \leftarrow \mathbf{y}$ 
2: while  $t \leq t_{\max}$  and  $\|\mathbf{r}\|_2 > \varepsilon \|\mathbf{y}\|_2$  do
3:    $\mathbf{s} \leftarrow \mathbf{A}^T \mathbf{r}$ 
4:    $i \leftarrow \arg \max_k |s_k|$ 
5:    $\mathbf{a} \leftarrow [A_{1,i}, A_{2,i}, \dots, A_{M,i}]^T$ 
6:    $\sigma \leftarrow s_i / \|\mathbf{a}\|_2^2$ 
7:    $\hat{x}_i \leftarrow \hat{x}_i + \sigma$ 
8:    $\mathbf{r} \leftarrow \mathbf{r} - \sigma \mathbf{a}$ 
9:    $t \leftarrow t + 1$ 
10: end while
11: return  $\hat{\mathbf{x}}$ 
```

3.2.2 Implementacja w języku MATLAB



Za pomocą załączonego skryptu MATLAB-owskiego⁵ `raindemo.m` o następującej zawartości:

```

%% Odtwarzanie sygnału kompresywnie spróbkowanego metodą
%% kroczącego dopasowania (Matching Pursuit)

%% Definicja rozmiarów problemu
N = 365; % długość oryginalnego sygnału
K = 20; % liczba niezerowych próbek sygnału
M = 50; % liczba obserwacji przetworzonego sygnału
```

⁵Program ten można także uruchomić w darmowym środowisku GNU Octave, co pozwala na wykonanie niniejszego podpunktu w domu.

```

%% Generacja oryginalnego sygnału do odtworzenia
x = zeros(N, 1); % rezerwacja miejsca na oryginalny sygnał
ix = randperm(N); % losowy wybór indeksów próbek sygnału
x(ix(1:K)) = abs(randn(K, 1)); % wstawienie losowych wartości

%% Generacja losowej macierzy pomiarowej
tau = 30; % stała czasowa zaniku sygnału obserwowanego
Ap = toeplitz(exp(-(0:N-1)/tau), [1, zeros(1, N-1)]);
iA = randperm(N); % losowy wybór indeksów pomiarów
A = Ap(iA(1:M), :); % macierz (losowo wybranych) pomiarów
y = A * x; % wektor obserwacji

%% Algorytm kroczącego dopasowania (Matching Pursuit)
xr = zeros(N, 1); % zerowanie miejsca na odtworzony sygnał
r = y; % początkowo "pozostałością" jest sygnał obserwowany
nrm2y = norm(y);
nrm2r = nrm2y;
t = 1; % licznik iteracji alorytmu
while nrm2r > 0.05 * nrm2y & t <= 50 % główna pętla algorytmu
    sp = A.' * r; % obliczenie wektora iloczynów skalarnych
    [dummy, i] = max(abs(sp)); % znalezienie maks. z nich,
    nrm2a = norm(A(:, i)); % norma odpowiedniej kolumny A
    s = sp(i); % pobranie maksymalnego iloczynu skalarnego
    s = s / nrm2a^2; % unormowanie udziału w "pozostałości"
    xr(i) = xr(i) + s; % wstawienie udziału do odtwarzanego x
    r = r - s * A(:, i); % aktualizacja wektora "pozostałości"
    nrm2r = norm(r); % obliczenie normy "pozostałości"
    fprintf('iter.%3d: x(%3d) <- %4.2f, nrm2res=%4.2f\n', ...
        t, i, s, nrm2r); % wydruk kontrolny
    t = t + 1; % aktualizacja licznika iteracji
end

%% Graficzna prezentacja wyników działania algorytmu
Apx = Ap * x; % wszystkie próbki sygnału obserwowanego
subplot(311); plot(x);
    ylabel('sygn. oryg.');
```

```
fwrite(f, xr, 'single');
fclose(f);
```

- Po zdefiniowaniu rozmiarów problemu generujemy oryginalny sygnał x do odtworzenia.
- Generujemy losową macierz pomiarową A .
- Za pomocą algorytmu kroczącego dopasowania odtwarzamy sygnał \hat{x} .
- Przedstawiamy na jednym rysunku sygnał oryginalny x , pełny sygnał obserwowany y_p z zaznaczonymi (zielonymi gwiazdkami) punktami losowego jego próbkowania.
- Zapamiętujemy w pliku dyskowym `raindemo.dat` rozmiary macierzy A jako liczby całkowite, a potem kolejno: wektor x , macierz pomiaru A , wektor obserwacji y i (dla porównania z naszymi wynikami) odtworzony za pomocą MATLAB-a wektor \hat{x} jako liczby zmiennoprzecinkowe pojedynczej precyzji.

W tym celu wykonujemy w okienku komend MATLAB-a polecenie:

```
raindemo
```

Warto je wykonać kilka razy, aż uzyskamy sytuację, gdy norma sygnału błędu będzie mała (wyraźnie mniejsza od jedności), a najniższy i najwyższy wykres będą jak najbardziej zbliżone do siebie. Dalej będziemy pracować korzystając z zapamiętanych na dysku wektorów i macierzy.

3.2.3 Implementacja w języku CUDA C z wykorzystaniem biblioteki CUBLAS



1. Tworzymy plik oprogramowania szkieletowego `raindemo.c` o następującej zawartości:

```
/* Odtwarzanie sygnału kompresywnie spróbkowanego */

#define _CRT_SECURE_NO_WARNINGS
#define WINDOWS_LEAN_AND_MEAN
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "raindemo.h"

/*****/

void alloc_mem(int M, int N,
               float ** x_ptr, float ** A_ptr, float **y_ptr,
               float **xr_ptr, float **xs_ptr)
{
```

```

* x_ptr = (float *) malloc(    N * sizeof(float));
* A_ptr = (float *) malloc(M * N * sizeof(float));
* y_ptr = (float *) malloc(M      * sizeof(float));
*xr_ptr = (float *) malloc(    N * sizeof(float));
*xs_ptr = (float *) calloc(    N , sizeof(float));
}

void free_mem(float *x,  float *A, float *y,
              float *xr, float *xs)
{
    free(x);
    free(A);
    free(y);
    free(xr);
    free(xs);
}

void read_data(int      * M_ptr, int      * N_ptr,
               float ** x_ptr, float ** A_ptr, float **y_ptr,
               float **xr_ptr, float **xs_ptr)
{
    FILE *f = fopen("raindemo.dat", "rb");

    fread(M_ptr, sizeof(int), 1, f);
    fread(N_ptr, sizeof(int), 1, f);

    alloc_mem(*M_ptr, *N_ptr,
              x_ptr, A_ptr, y_ptr, xr_ptr, xs_ptr);

    fread(* x_ptr, sizeof(float),      *N_ptr, f);
    fread(* A_ptr, sizeof(float), *M_ptr * *N_ptr, f);
    fread(* y_ptr, sizeof(float), *M_ptr,      f);
    fread(*xr_ptr, sizeof(float),      *N_ptr, f);

    fclose(f);
}

/*****/

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

```

```

/*****
int main(int argc, char *argv[])
{
    int      M,  N;
    float    *x, *A, *y, *xr, *xs;
    read_data(&M, &N, &x, &A, &y, &xr, &xs);
    csmpl(xs, y, A, N, M, argc, argv);
    sigcmp(xs, xr, N);
    free_mem(x, A, y, xr, xs);
    if (IsDebuggerPresent()) getchar();
    return 0;
}

```

Podobnie jak w poprzednim ćwiczeniu, oprogramowanie szkieletowe jest bardzo podobne do szczegółowo omawianego na wykładzie programu mnożenia macierzy i dlatego nie będzie tu dokładnie opisywane.

Kod ten jest napisany bardzo minimalistycznie, aby nie zaciemniać struktury programu. Pominięto w nim nawet pomiar czasu wykonania programu. Część szkieletowa kodu nie będzie ulegała zmianie – tworzyć będziemy tylko zawarty w pliku `raindemo.cu` kod „wykonawczy”.

2. Interfejs pomiędzy kodem szkieletowym a wykonawczym jest opisany w pliku `raindemo.h` o następującej zawartości:

```

#ifdef __cplusplus
extern "C"
#endif
void csmpl(float *xs, float *y, float *A, int N, int M, ...);

```

3. Tym razem także nie będziemy tworzyli implementacji odniesienia na CPU, tylko zacięniemy od szkieletu wersji na procesor graficzny. Jej źródło jest zawarte w pliku `raindemo.cu` o następującej zawartości:

```

#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <helper_cuda.h> // -I$(NVCUDASAMPLES_ROOT)/common/inc
#include <cublas_v2.h>    // wymaga konsolidacji z cublas.lib

#include <stdio.h>

#include "raindemo.h"

#define TRY(code) checkCudaErrors((cudaError_t) (code))

void csmpl(float *xs, float *y, float *A, int N, int M, ...)
{
    float *dev_A, *dev_r, *dev_sp;
    cublasHandle_t h;
    float one = 1.0f, zero = 0.0f, nrm2y, nrm2a, nrm2r, s;

```



```

int i, t;

TRY(cudaSetDevice(0));

TRY(cudaMalloc(&dev_A, M*N*sizeof(float)));
TRY(cudaMalloc(&dev_r, M *sizeof(float)));
TRY(cudaMalloc(&dev_sp, N*sizeof(float)));

TRY(cublasCreate(&h));

TRY(cublasSetMatrix(M, N, sizeof(float), A, M, dev_A, M));
TRY(cublasSetVector(M, sizeof(float), y, 1, dev_r, 1));
TRY(cublasSnrm2(h, M, dev_r, 1, &nrm2y));

for (t=1, nrm2r=nrm2y; t<=50 && nrm2r>0.05*nrm2y; t++)
{
    // TODO: iteracja algorytmu

    printf("iter.%3d: x(%3d) <- %4.2f, nrm2res=%4.2f\n",
           t, i, s, nrm2r);
}

TRY(cublasDestroy(h));

TRY(cudaFree(dev_sp));
TRY(cudaFree(dev_r));
TRY(cudaFree(dev_A));

TRY(cudaDeviceReset()); // dla debuggera i profilera
}

```

Struktura tego pliku jest dość prosta i typowa. Na wstępie zwróćmy tylko uwagę, że zamiast korzystać z makrodefinicji `checkCudaErrors()`, zastąpimy ją własną definicją `TRY{}`, po to, aby móc w jednolity sposób „opakowywać” wywołania funkcji `cudaXxxx()` i `cublasXxxx()`. Funkcja wykonawcza nazywa się `csm`, co jest skrótem od *Compressive Sensing Matching Pursuit*.

Dokładniejszego omówienia wymagają zadeklarowane w programie zmienne. Wskaźniki `dev_A`, `dev_r`, `dev_sp` odpowiadają w pamięci urządzeniom macierzy A oraz wektorom r i s . Stałe `one` i `zero` będą potrzebne przy wywoływaniu funkcji biblioteki CUBLAS jako stałe α i β . Zmienne `nrm2y`, `nrm2a`, `nrm2r` odpowiadają $\|y\|_2$, $\|a\|_2$, $\|r\|_2$ z opisu algorytmu, zaś zmienna `s` – współczynnikowi σ . Znaczenie zmiennych całkowitych `i` oraz `t` jest identyczne jak i oraz t z opisu algorytmu.

4. Pamiętajmy, aby we właściwościach projektu do opcji *Configuration Properties* — *VC++ Directories* — *Include Directories* dodać pozycję `$(NVCUDASAMPLES_ROOT)/common/inc` dla wszystkich konfiguracji projektu. Jest ona niezbędna do znalezienia pliku `helper_cuda.h`. Pamiętajmy także, aby nasz projekt był 64-bitowy.
5. Ponadto we właściwościach projektu do opcji *Configuration Properties* — *Linker* —

Input — Additional Dependencies dodajemy pozycję `cublas.lib` dla wszystkich konfiguracji projektu, aby prawidłowo skonsolidować nasz projekt z biblioteką CUBLAS.

6. Na podstawie pseudokodu algorytmu oraz jego implementacji w języku MATLAB uzupełniamy wnętrze pętli algorytmu kroczącego dopasowania. Prolog został już uzupełniony i może stanowić wzór dla uzupełnianego kodu. W prologu pętli wykorzystano funkcje biblioteki CUBLAS:

- (a) `cublasSetMatrix()` – do przepisania macierzy A do `dev_A`. Wiodącym rozmiarem (*leading dimension*) obu tych macierzy jest ich liczba wierszy, czyli M .
- (b) `cublasSetVector()` – do przepisania wektora y do `dev_r`. Ponieważ przepisujemy kolejne elementy, więc jako przyrosty indeksów (*increment*) obu wektorów podajemy jedyńki.
- (c) `cublasSnrm2()` – do policzenia $\|y\|_2$ dla potrzeb sprawdzania warunku końca pętli. I tutaj liczymy normę kolejnych elementów wektora, więc ustawiamy przyrost indeksu równy 1. Funkcja zwraca status błędu, a obliczoną normę wstawia do przekazywanego przez adres ostatniego parametru wywołania.

Wewnątrz pętli należy kolejno użyć następujących funkcji biblioteki CUBLAS:

- (a) `cublasSgemv()` – w kroku 3 algorytmu. Ponieważ mnożymy macierz A^T , więc jako operację nad macierzą trzeba podać transpozycję (`CUBLAS_OP_T`). Jako parametry α i β trzeba podać adresy zmiennych odpowiednio `one` i `zero`, a nie literały `1.0f` czy `0.0f`. Po raz kolejny wiodącym rozmiarem jest M , a przyrosty indeksów obu wektorów powinny wynosić 1.
- (b) `cublasIsamax()` – w kroku 4 algorytmu. Funkcja zwraca status błędu, a obliczony indeks wstawia do przekazywanego przez adres ostatniego parametru wywołania. Uwaga: elementy są, zgodnie z konwencją FORTRAN-owską, indeksowane od *jedności*, a nie od zera.
- (c) `cublasSnrm2()` – w kroku 6 algorytmu. Liczymy tu normę wektora, będącego i -tą kolumną macierzy A . Ponieważ biblioteka CUBLAS, zgodnie z konwencją FORTRAN-owską, przechowuje macierze *kolumnami*, więc nasz wektor będą tworzyły kolejne elementy macierzy. Jako adres wektora należy zatem podać adres pierwszego elementu i -tej kolumny macierzy (raz jeszcze przypomnijmy, że kolumny numerowane są od jedności!), a jako przyrost indeksu – jedyńkę.
- (d) `cublasGetVector()` – do przepisania jednej liczby σ do pamięci gospodarza. Choć przepisujemy tylko jedną liczbę (i -ty element wektora do skłara), to jako przyrosty obu indeksów musimy podać jedyńki. No i po raz kolejny przypomnijmy, że elementy wektora numerowane są od jedności!
- (e) `cublasSaxpy()` – w kroku 8 algorytmu. Już sama nazwa funkcji podkreśla, że jest to αx **plus** y . A my musimy zrobić **minus**. Można tymczasowo zmienić znak zmiennej s , której adres przekazujemy do funkcji, albo wprowadzić nową zmienną pomocniczą.
- (f) `cublasSnrm2()` – przygotowujemy się do sprawdzenia warunku końca pętli, weryfikowanego w kroku 2 algorytmu. Normę liczyliśmy już dwa razy – za trzecim razem na pewno damy sobie radę bez zbędnych wskazówek.

7. Po napisaniu brakującego kodu uruchamiamy program, sprawdzając, czy podawana na końcu jego pracy wartość maksymalnego bezwzględnego błędu (pomiędzy wektorami odtworzonymi przez MATLAB-a i przez nasz program) jest dostatecznie mała. Jeśli tak nie jest, to warto prześledzić wydruki kontrolne wyprowadzane w każdym obiegu pętli przez wersję CUDA C i porównać je z wydrukami wyprowadzanymi przez program MATLAB-owski.

Rozdział 4

Implementacja operacji numerycznych w układzie FPGA

Celem ćwiczenia jest badane różnych sposoby implementacji podstawowych operacji arytmetycznych, tzn. dodawania i mnożenia w układach FPGA. Jako przykład wykorzystania tych operacji posłuży dobrze już znane zagadnienie filtracji sygnałów, realizowane w ramach drugiego ćwiczenia na platformie CUDA.

Ćwiczenie realizowane jest w środowisku projektowym *Vivado* z wykorzystaniem płytki ewaluacyjnej z układem FPGA: *Basys 3* firmy Digilent.

Podczas laboratorium należy wypełniać formatkę sprawozdania, odpowiadając krótko i konkretnie na postawione pytania. Należy zgłaszać prowadzącemu każde wykonane zadanie i możliwie szybko zgłaszać ew. trudności w jego wykonaniu. Przed laboratorium należy wykonać w domu zadanie [4.4.1](#).

4.1 Płytką ewaluacyjną Basys 3

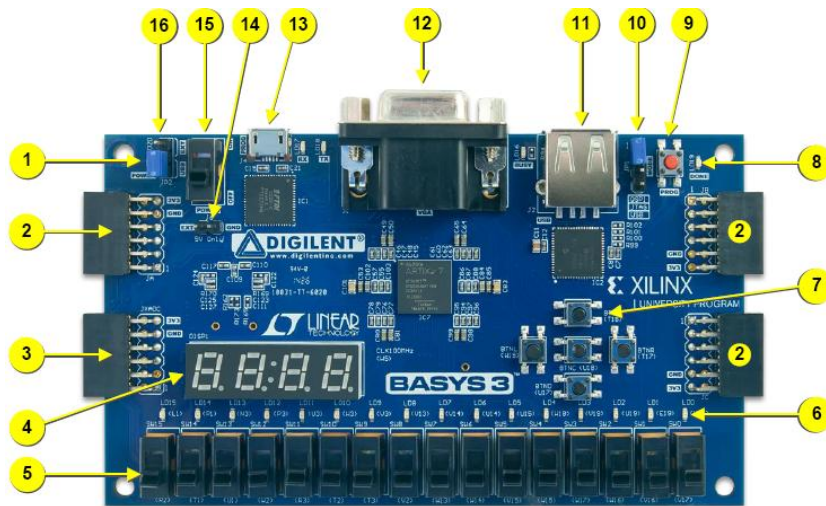
Płytką *Basys 3* (pokazana na rys. [4.1](#)), w którą wyposażone są stanowiska laboratoryjne, zawiera układ FPGA typu XC7A35T w obudowie CPG236, pochodzący z rodziny Artix-7 firmy AMD/Xilinx. Najważniejsze dla nas zasoby tego układu zestawiono w tab. [4.1](#).

Tabela 4.1: Dostępne zasoby układu XC7A35T

Liczba komórek logicznych	33280
Liczba komórek SLICE	5200
Pojemność pamięci Block RAM	1800 Kbit
Liczba bloków obliczeniowych DSP	90
Max. częstotliwość taktowania	450 MHz

Układ FPGA, oprócz niezbędnych obwodów zasilania, otoczony jest układami peryferyjnymi. Niektóre z nich będą przydatne podczas realizacji ćwiczeń laboratoryjnych oraz projektów na platformie FPGA. Do najważniejszych peryferii należą (odwołania w nawiasach dotyczą rys. [4.1](#)):

- interfejs USB (13) z kontrolerem JTAG i mostkiem portu szeregowego UART,



Rysunek 4.1: Wygląd płyty ewaluacyjnej Basys-3

- przyciski, przełączniki, diody LED (5, 6, 7),
- pamięć konfiguracyjna QSPI FLASH,
- interfejs USB-Host do dołączenia urządzeń typu HID (11),
- przetwornik XADC wbudowany do układu FPGA (3),
- prosty interfejs do podłączenia monitora VGA (12),
- wyświetlacze siedmiosegmentowe LED (4).

Na komputerach w laboratorium w katalogu C:\RIM\FPGA\Basys-3\ znajduje się kopia najważniejszych plików z dokumentacji płyty (instrukcja użytkownika, schemat ideowy, itp.) Pełna dokumentacja płyty jest dostępna wraz z przykładowymi projektami w Internecie pod adresem: <https://digilent.com/reference/programmable-logic/basys-3/start>.

Podczas ćwiczeń laboratoryjnych płyta uruchomieniowa powinna być dołączona do komputera PC za pomocą kabla mikro-USB. Napięcie zasilania (5V) domyślnie pobierane jest z portu USB (ozn. 13 na rys. 4.1). Włącznik zasilania (SW16, ozn. 15 na rys. 4.1), umieszczony w lewym górnym rogu płyty, powinien znajdować się w pozycji „ON”.

Po włączeniu zasilania świeci się czerwona dioda LD20 (POWER, ozn. 1 na rys. 4.1) i automatycznie ładowana jest z pamięci *FLASH* pierwsza konfiguracja układu FPGA zawierająca przykładowy projekt. Po skonfigurowaniu układu FPGA świeci się zielona dioda LD19 (DONE, ozn. 8 na rys. 4.1). Działanie tej konfiguracji sygnalizowane jest miganiem diody LD6 i „wędrującym światłem” na diodach LD8...LD15 na płycie uruchomieniowej. Na tym etapie możliwa jest już komunikacja pomiędzy płytą a komputerem PC (patrz p. 4.2), a w układzie FPGA zaprogramowany jest trywialny algorytm „przetwarzania danych”, opisany w p. 4.3.2).

Domyślne ustawienia zworek (*jumper’ów*) na płytce Basys-3: JP1:QSPI, JP2:USB, jak pokazano na rys. 4.1.

4.2 Komunikacja płyty z komputerem PC

Komunikacja z komputerem odbywa się przez port USB. Połączenie to realizuje jednocześnie dwie funkcje:

- interfejs JTAG – ładowanie konfiguracji układu FPGA,
- interfejs UART – komunikacja z algorytmem uruchomionym w układzie FPGA.

Komunikacja z algorytmem uruchomionym w FPGA umożliwia:

- przesłanie danych wejściowych z komputera do algorytmu,
- uruchomienie pracy i kontrolę algorytmu,
- dokładny pomiar czasu pracy algorytmu,
- przesłanie danych wynikowych do komputera.

Po stronie komputera PC komunikacja jest realizowana przez skrypt `fpga.m` uruchamiany w środowisku Matlab. Aby komunikacja była możliwa, po drugiej stronie (tzn. w projekcie załadowanym do układu FPGA) musi znajdować się blok `rimlab01`, opisany dalej.

4.2.1 Komunikacja przez USB – interfejs JTAG

Dla potrzeb ćwiczenia, ładowanie konfiguracji do układu FPGA odbywać się będzie wyłącznie przez interfejs JTAG. Jest on zaimplementowany na płycie uruchomieniowej z wykorzystaniem układu FTDI, który jest połączony z portem USB komputera za pomocą przewodu z wtykiem micro USB.

Na komputerze PC, do konfigurowania układu FPGA wykorzystywany będzie moduł *Hardware Manager*, będący składnikiem środowiska *Vivado* (patrz p. 4.2.5). W ten sposób możliwe jest załadowanie własnego projektu do pamięci konfiguracyjnej układu FPGA. W ramach ćwiczeń nie będziemy programować pamięci konfiguracyjnej FLASH.

4.2.2 Blok komunikacji: `rimlab01`

Blok `rimlab01`, dostępny jako zestaw źródeł VHDL, realizuje funkcje komunikacji przez port szeregowy (UART) w protokole zgodnym ze skryptem `fpga.m`, uruchamianym na komputerze PC. Deklaracja tego komponentu, która musi znaleźć się w kompilowanym podczas ćwiczenia pliku VHDL, ma następującą postać:

```
component rimlab01 is
  Port (
    -- Komunikacja ze światem zewnętrznym
    CLK : in  STD_LOGIC;
    RST : in  STD_LOGIC;
    RXD : in  STD_LOGIC;
    TXD : out STD_LOGIC;

    -- Komunikacja z algorytmem przetwarzania danych
```

```

START_PROC    : out STD_LOGIC;
PROC_WORKING  : in  STD_LOGIC;

DOUT_DATA     : out STD_LOGIC_VECTOR (15 downto 0);
RD_DATA       : in  STD_LOGIC;

DIN_RESULT    : in  STD_LOGIC_VECTOR (15 downto 0);
WR_RESULT     : in  STD_LOGIC
);
end component;

```

Przeznaczenie poszczególnych portów komponentu opisano w tab. 4.2. Wszystkie sygnały (z wyjątkiem linii RXD, TXD) są aktywne w stanie wysokim.

Na rys. 4.2 pokazano schemat połączeń komponentu z jego otoczeniem wewnątrz układu FPGA, zaś w tab. 4.3 podsumowano zajętość zasobów FPGA dla tego komponentu. Na płycie uruchomieniowej linie RXD, TXD są połączone z układem konwertera UART/USB firmy FTDI. Do wejścia CLK dołączony jest generator kwarcowy o częstotliwości 100 MHz. Wejście RST układu należy natomiast dołączyć do wejścia połączonego z przyciskiem wymuszającym stan wysoki po wciśnięciu, np. z portem btnC, tak jak w przykładowym projekcie `fpgatop.vhd` (patrz p. 4.3.2).

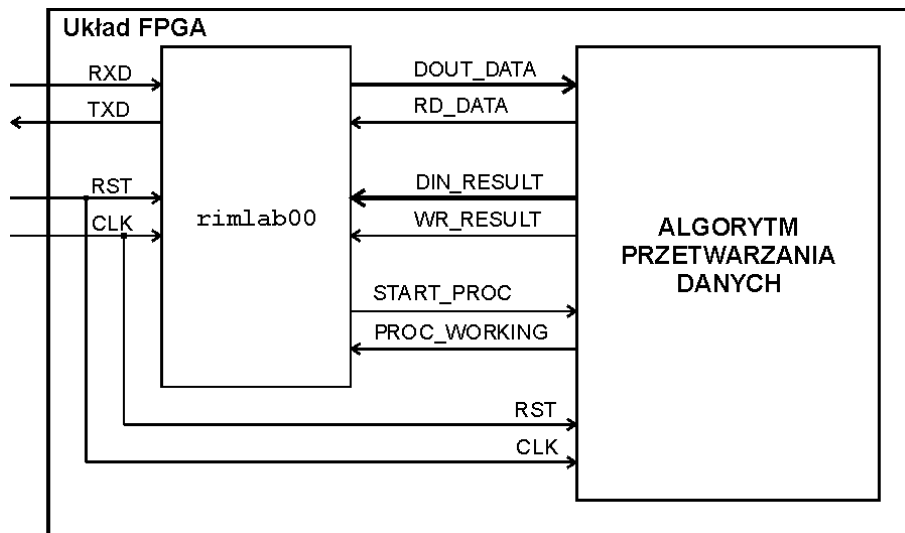
Połączenie sygnałów logicznych z fizycznymi wyprowadzeniami układu (IOB) jest realizowane przez odpowiednie wpisy do pliku `.XDC` dołączonego do projektu (patrz p. 4.2.3).

Tabela 4.2: Przeznaczenie portów komponentu rimlab01

Nazwa portu	Opis portu
CLK	Wejście zegara taktującego (100 MHz)
RST	Wejście sygnału zerującego komponent
RXD	Szeregowe wejście danych (UART)
TXD	Szeregowe wyjście danych (UART)
START_PROC	Wyjście sygnału uruchamiającego pracę algorytmu obliczeniowego
PROC_WORKING	Wejście flagi sygnalizującej pracę algorytmu
DOUT_DATA	Wyjście 16-bitowych danych źródłowych dla algorytmu
RD_DATA	Sygnał odczytujący kolejne słowa danych z DOUT_DATA
DIN_RESULT	Wejście 16-bitowych danych wynikowych z algorytmu
WR_RESULT	Sygnał wpisujący kolejne słowo wyniku do DIN_RESULT

Tabela 4.3: Zasoby zajmowane przez komponent rimlab01

Przerzutniki (FF, <i>Flip Flop</i>)	239
Tablice LUT	292
Komórki SLICE	115
Bloki pamięci RAMB18E1	2
Porty I/O (IOB)	4



Rysunek 4.2: Schemat połączeń algorytmu z komponentem rimlab01

4.2.3 Przykładowy plik ograniczeń (.XDC)

W pliku `basys-rim.xdc` zapisane są m.in. numery wykorzystywanych wyprowadzeń układu FPGA, połączone na płycie uruchomieniowej z odpowiednimi peryferiami:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5    IOSTANDARD LVCMOS33 } [
    get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [
    get_ports clk] ...

## Switches
set_property -dict { PACKAGE_PIN V17    IOSTANDARD LVCMOS33 } [
    get_ports {sw0}]
set_property -dict { PACKAGE_PIN V16    IOSTANDARD LVCMOS33 } [
    get_ports {sw1}]
set_property -dict { PACKAGE_PIN W16    IOSTANDARD LVCMOS33 } [
    get_ports {sw2}]
# ...

## LEDs
set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 } [
    get_ports {led0}]
set_property -dict { PACKAGE_PIN E19    IOSTANDARD LVCMOS33 } [
    get_ports {led1}]
# ...

##Buttons
set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 } [
    get_ports btnC]
set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 } [
    get_ports btnU]
# ...

##USB-UART Interface
```



```
set_property -dict { PACKAGE_PIN B18      IOSTANDARD LVCMOS33 } [
    get_ports rxd]
set_property -dict { PACKAGE_PIN A18      IOSTANDARD LVCMOS33 } [
    get_ports txd]
```

Aby skompilowany podczas ćwiczenia projekt miał szansę zadziałać na platformie sprzętowej, do projektu musi być dołączony wspomniany plik `.XDC`. Zapewnia on wyprowadzenie sygnałów logicznych na właściwe „piny” układu FPGA (bloki IOB), fizycznie połączone z peryferiami na płycie uruchomieniowej.

W przypadku braku pliku `.XDC` w projekcie, sygnały zostały by wyprowadzone na przypadkowe bloki IOB i układ FPGA nie mógłby współpracować ze swoim otoczeniem na płycie uruchomieniowej.

W pliku `.XDC` powinny się znajdować wyłącznie odwołania do portów występujących w kompilowanej jednostce projektowej (`ENTITY`) – wszelkie niepotrzebne deklaracje należy zakomentować znakiem `#`. W przypadku wystąpienia dwołania do portu nie zadeklarowanego w liście portów kompilowanej jednostki, lub zadeklarowanego, ale nie połączonego z blokami logicznymi, środowisko projektowe zgłosi błąd i przerwie procedurę implementacji projektu.

4.2.4 Komunikacja w środowisku Matlab: skrypt `fpga.m`

Jeśli układ FPGA ma załadowaną odpowiednią konfigurację (np. po włączeniu zasilania – z pamięci FLASH) i płyta uruchomieniowa jest przyłączona do komputera przez port UART/USB, można sprawdzić komunikację, uruchamiając funkcję `fpga()` w środowisku Matlab.

Argumentem funkcji `fpga()` jest wektor danych wejściowych, złożonych z liczb rzeczywistych z zakresu $-32768 \dots 32767$, o maksymalnej długości 1024 słów. Wektor ten jest przesyłany (z pominięciem części ułamkowych) do bloku pamięci RAM w układzie FPGA. W przypadku podania wektora danych krótszego niż 1024 liczby, zostanie on uzupełniony zerami do pełnej długości bufora. Po załadowaniu danych wejściowych uruchamiany jest algorytm przetwarzania, zdefiniowany przez użytkownika.

Podczas pracy algorytm wpisuje swoje wyniki obliczeń (16-bitowe liczby całkowite ze znakiem) do drugiego bufora w pamięci RAM o pojemności 1024 słów 16-bitowych. Po zakończeniu pracy algorytmu pełna zawartość tego bufora jest zwracana przez funkcję `fpga()` do środowiska Matlab.

Przykład wywołania funkcji z danymi wejściowymi (przygotowanymi w wektorze `x`) zamieszczono poniżej:

```
x=32767*(sin((0:1023)*2*pi/100))';
y=fpga(x);
```

Funkcja `fpga()` wyświetla również inne komunikaty związane z działaniem algorytmu, np. czas pracy netto (liczony w cyklach zegarowych i w milisekundach), czy algorytm prawidłowo zakończył działanie, itp:

```
>>y=fpga(x);
Algorytm przetwarzania w FPGA zakończył pracę.
Liczba odebranych słów wyniku : 1024
Czas przetwarzania : 0.06146 ms
Liczba cykli zegara na przetwarzanie : 3073
Liczba cykli zegara na słowo wyniku : 3.001
Odczytano wyniki przetwarzania z układu FPGA.
```

Jeśli komunikacja przebiegła prawidłowo i algorytm obliczeniowy wykonał swoją pracę, to w wektorze *y* zwróconym przez funkcję będą zapisane dane wynikowe, odczytane z bufora w układzie FPGA. Dane wejściowe oraz wynikowe można zestawić na wspólnym wykresie:

```
t=0:1023;
plot (t, x, t, y);
```

Wszystkie powyższe komendy testujące komunikację można również uruchomić, wywołując załączony skrypt `testcom.m`.

4.2.5 Środowisko projektowe Vivado

Środowisko *AMD/Xilinx Vivado* stanowi kompletne narzędzie do realizacji projektów w układach FPGA firmy Xilinx (m.in. z rodzin: Spartan-7, Artix-7, Kintex-7, Virtex-7, Zynq-7000, oraz pozostałych nowszych układów). Instalator środowiska (z bezpłatną licencją *WebPACK*) można pobrać ze strony: <https://www.xilinx.com/support/download.html>, po uprzednim założeniu konta i zalogowaniu się.

Na komputerach w laboratorium zainstalowano wersję 2018.2 środowiska *Vivado*. Pakiet ten umożliwia tworzenie projektów dla większości niskobudżetowych układów FPGA z serii ”-7” firmy AMD/Xilinx. Większe układy FPGA są wspierane w tym środowisku wyłącznie komercyjną licencją.

W skład pakietu *Vivado* wchodzi m.in. następujące narzędzia:

- środowisko projektowania w języku VHDL, Verilog,
- edytor schematów blokowych (IP Integrator),
- narzędzie do syntezy logicznej (kompilator języków HDL),
- symulator logiczny,
- analizator stanów logicznych (debugowanie projektu z blokami ILA)
- generator kompletnych bloków funkcjonalnych (bloków IP): *IP Catalog*,
- moduł do konfigurowania układów FPGA: *Hardware Manager*.

4.3 Pierwszy projekt w języku VHDL dla układu FPGA

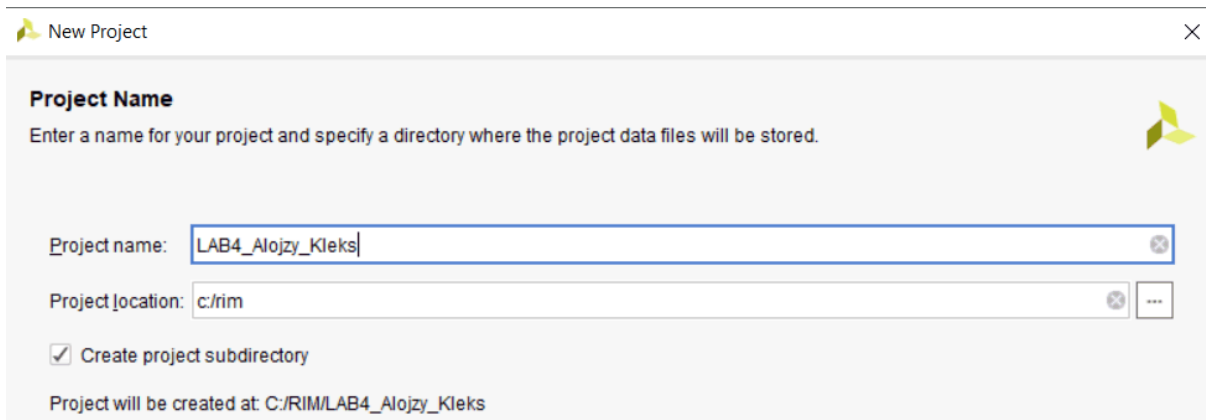
4.3.1 Tworzenie projektu i dodanie źródeł

Dokonamy kompilacji i uruchomienia pierwszego projektu w języku VHDL przeznaczonego dla układu FPGA na płycie ewaluacyjnej *Basys 3*. Na początku zajęć należy uruchomić system operacyjny *Windows* na komputerze dostępnym na stanowisku. Na pulpicie należy odszukać i uruchomić ikonę *Vivado 2018.2* (nie pomylić z *Vivado HLS 2018*, która uruchamia inne środowisko).

Po uruchomieniu środowiska stworzymy nowy projekt za pomocą kreatora. W tym celu należy:

1. z okna *Quick Start* wybrać *Create Project*,

2. w nowym oknie *New Project* kliknąć na przycisku *Next*,
3. wpisać nazwę projektu *Project name*, np. `LAB4_Imię_Nazwisko` i ustawić jego lokalizację na dysku *Project location* w katalogu np. `c:\rim` – patrz rys. 4.3,



Rysunek 4.3: Wybór nazwy i lokalizacji projektu

4. w oknie *Project Type* wybrać: *RTL Project* (domyślne ustawienie), kliknąć *Next*,
5. w oknie *Add Sources*: kliknąć *Add Files*, przejść do folderu: `c:\rim\fpga\rtl`,
6. zaznaczyć wszystkie widoczne pliki z folderu, kliknąć na przycisku *OK* a następnie *Next*,
7. w oknie *Add Constraints* kliknąć *Add Files*, wskazać plik: `basys-rim.xdc`, kliknąć na przycisku *OK* a potem *Next*,
8. w oknie *Default Parts*, w zakładce *Parts* odnaleźć i z rodziny *Artix-7* wybrać układ: **xc7a35tcbg236-1** (patrz rys. 4.4), kliknąć na przycisku *Next*,
9. w oknie *New Project Summary* kliknąć na przycisku *Finish*.

Po wykonaniu tych operacji środowisko Vivado przez kilka sekund analizuje pliki i tworzy nowy projekt.

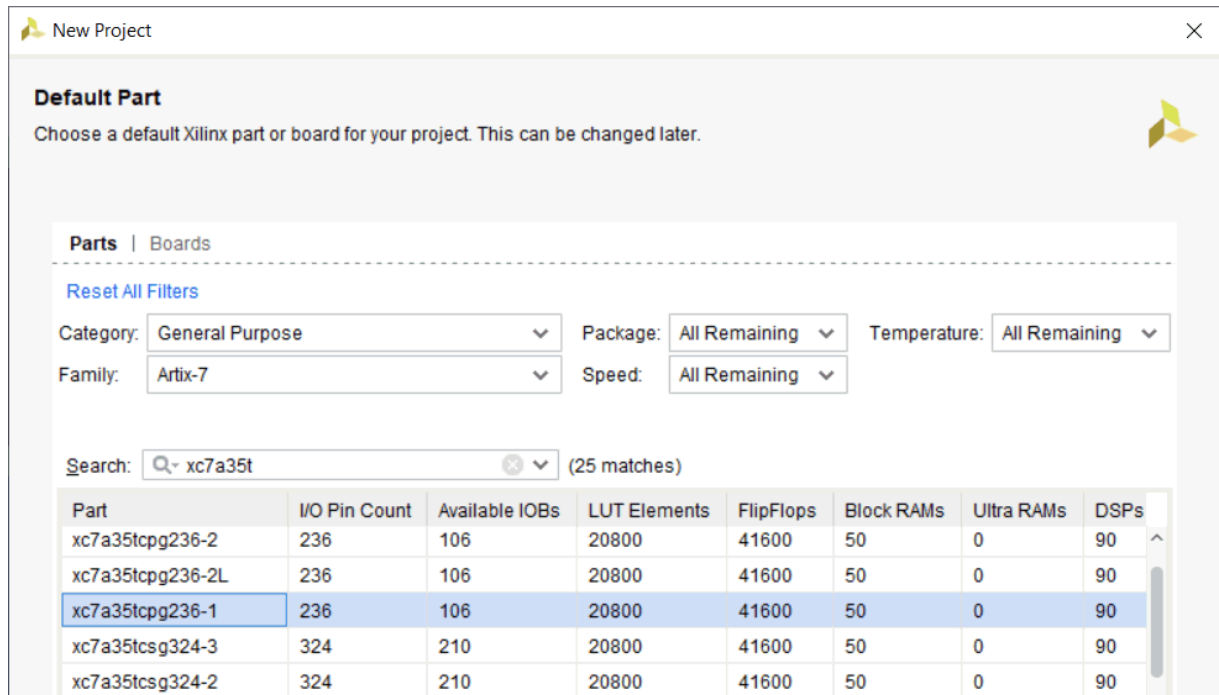
4.3.2 Główny plik projektu – `fpgatop.vhd`

W pliku `fpgatop.vhd` utworzony jest przykładowy działający projekt dla układu FPGA. Projekt składa się dwóch części: modułu komunikacji `rimlab01` (opisanego w p. 4.2.2) oraz prymitywnego „algorytmu przetwarzania danych”.

Algorytm przetwarzania danych początkowo znajduje się w stanie bezczynności (`IDLE`). Działanie algorytmu jest uruchamiane sygnałem `START` generowanym w module komunikacji. Po otrzymaniu tego sygnału, algorytm ustawia flagę `BUSY`, sygnalizującą przetwarzanie danych, i rozpoczyna pracę, wchodząc do stanu `RD_DAT`.

W algorytmie kopiowane są kolejne słowa danych wejściowych z podziałem liczby przez 2 lub z odwróceniem znaku, zależnie od stanu przełącznika `SW0` na płycie ewaluacyjnej. Funkcja ta jest zaimplementowana w dwóch wykonywanych na przemian stanach (`RD_DAT`, `WR_DAT`) automatu opisanego w procesie `fill_mem`:





Rysunek 4.4: Wybór układu FPGA do projektu

```

elsif stan_alg=RD_DAT then
    -- Odczytanie słowa z bufora i wykonywanie "przetwarzania"
    if sw0='0' then -- przełącznik wyłączony
        -- podział przez 2
        RESULT_DATA <= SRC_DATA(15) & SRC_DATA(15 downto 1);

    else -- przełącznik włączony
        RESULT_DATA <= X"0000"-SRC_DATA; -- odwracanie znaku
    end if;

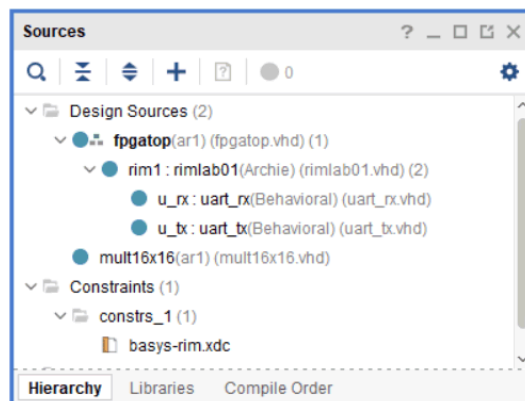
    SRC_RD <= '0';
    stan_alg<=WR_DAT;

elsif stan_alg=WR_DAT then
    -- Wpisanie słowa do bufora na dane wynikowe
    if RESULT_WR='1' then -- Dokonano wpisu słowa
        RESULT_WR<='0';
        if cnt_data=1023 then stan_alg<=IDLE;
            else cnt_data<=cnt_data+1;
                SRC_RD <= '1';
                stan_alg<=RD_DAT;
            end if;
        else RESULT_WR<='1'; -- wykonać wpis w nast. cyklu
        end if;
    
```

Pomocniczy licznik `cnt_data` zlicza słowa wpisane do bufora wyników. Po zapełnieniu całego bufora (1024 słowa) algorytm kończy pracę i powraca do stanu bezczynności (IDLE).

4.3.3 Kompilacja projektu

Jeśli wszystkie pliki źródłowe są już dodane do projektu, to w oknie pokazującym jego strukturę (*Hierarchy*) nie powinny pojawiać się znaki zapytania przy modułach. Moduł nadrzędny (tutaj: `fpgatop`) jest umieszczony na górze i oznaczony ikoną z trzema kwadracikami – patrz rys. 4.5.



Rysunek 4.5: Struktura źródeł projektu

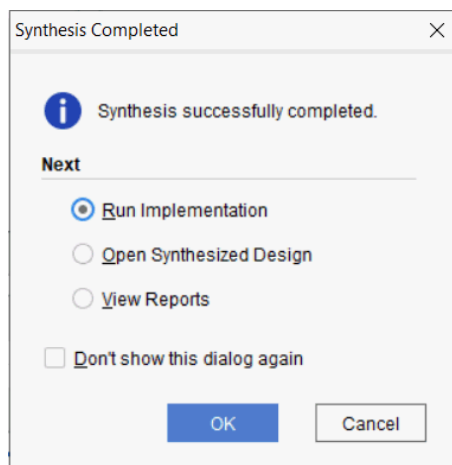
Można teraz uruchomić procedury syntezy logicznej i implementacji projektu. W tym celu po lewej stronie środowiska, w oknie *Flow Navigator* należy kliknąć na pozycji *Synthesis / Run Synthesis*. Spowoduje to uruchomienie syntezy logicznej.

1. W oknie *Launch Runs* nie zmieniamy żadnych ustawień, klikamy *OK*.
2. Po zakończeniu syntezy logicznej, w oknie *Design Runs* pokazywana jest wstępna zajętość zasobów (LUT, FF, BRAM, DSP – rys. 4.6). Jeśli proces syntezy przejdzie bez błędów, w oknie *Synthesis Completed* które się pojawi, zaznaczamy *Run implementation* i klikamy *OK* (patrz rys. 4.7).

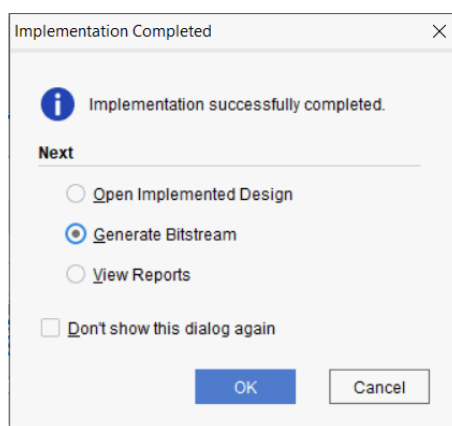
Design Runs															
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	
✓ synth_1	constrs_1	synth_design Complete!								338	2...	1.00	0	0	
▷ impl_1	constrs_1	Not started													

Rysunek 4.6: Zajętość zasobów po syntezie logicznej

3. W kolejnym oknie *Launch Runs* nie zmieniamy ustawień i klikamy *OK*. Implementację można także uruchomić niezależnie, klikając na pozycji *Implementation / Run Implementation* z okna *Flow Navigator*.
4. Aby wygenerować plik konfiguracyjny dla układu FPGA, w oknie *Implementation Completed* zaznaczamy pozycję *Generate Bitstream* i klikamy *OK* (rys. 4.8).
5. Alternatywnie, generację konfiguracji można także uruchomić, klikając na pozycji *Program and DEBUG / Generate Bitstream* z okna *Flow Navigator*. W kolejnym oknie *Launch Runs* nie zmieniamy ustawień i klikamy *OK*.



Rysunek 4.7: Okno ukończenia syntezy logicznej



Rysunek 4.8: Okno ukończenia implementacji

6. Po zakończeniu tej operacji, w polu *Design Runs* (na samym dole okna środowiska Vivado), w wierszu *impl_1* powinien pojawić się status: *write_bitstream Complete* oraz końcowe wskazanie zajętości zasobów FPGA (LUT, FF, BRAM, DSP – patrz rys. 4.9) wraz z czasem trwania poszczególnych procesów (*Design Runs / Elapsed*).

Tcl Console															Messages															Log															Reports															Design Runs															x																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													
Q															≡															⚙															⏮															⏪															⏩															⏭															+															%																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																
Name															Constraints															Status															WNS															TNS															WHS															THS															TPWS															Total Power															Failed Routes															LUT															FF															BRAMs															URAM															DSP																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						
▼															✓ synth_1															constrs_1															synth_design Complete!																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																											

Rysunek 4.9: Zajętość zasobów po implementacji

W tym momencie projekt został w pełni skompilowany i konfiguracja układu FPGA jest gotowa do przesłania do płytki ewaluacyjnej.

4.3.4 Konfiguracja układu FPGA na płycie ewaluacyjnej

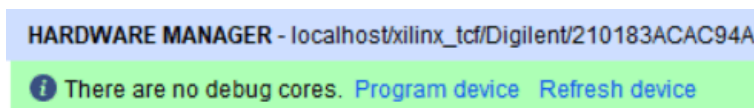
Po poprawnym przejściu wszystkich etapów syntezy logicznej i implementacji, w odpowiednim folderze projektu (*nazwa_projektu.runs/impl_1*) znajduje się plik *fpgatop.bit*

z konfiguracją do przesłania do układu FPGA.

Upewniamy się, że do portu USB komputera dołączona jest płytką Basys 3 i ma włączone zasilanie – czerwona dioda LED (*POWER*) powinna świecić.

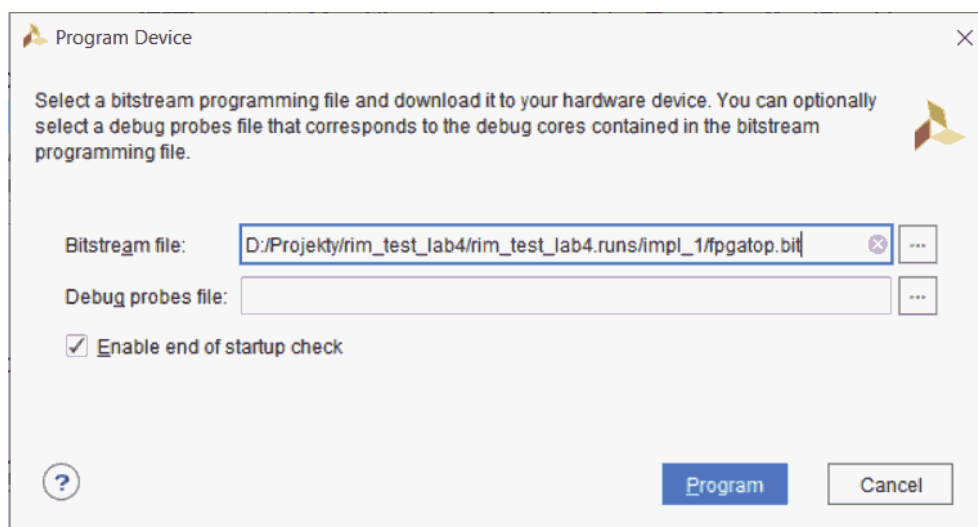
Aby przesłać nową konfigurację do układu FPGA, w środowisku Vivado:

1. Na samym dole okna *Flow Navigator* wyszukujemy pozycję *PROGRAM AND DEBUG / Open Hardware Manager* i klikamy na niej.
2. W górnej części okna Vivado pojawi się pasek *Hardware Manager - unconnected*, w którym klikamy na linku *Open target* i w menu kontekstowym które się pojawi, wybieramy pozycję: *Auto Connect*.
3. Po uzyskaniu połączenia JTAG z płytką Basys 3, w pasku *Hardware Manager* pojawi się informacja o nawiązaniu połączeniu z opcją (linkiem) *Program device*, na której klikamy (rys. 4.10).



Rysunek 4.10: Okno połączenia z układem FPGA

4. Pojawi się okno *Program Device*, ze wskazaniem pliku *fpgatop.bit* do zaprogramowania (patrz rys. 4.11). Nie zmieniając ustawień, klikamy na przycisku *Program*.



Rysunek 4.11: Wybór pliku do zaprogramowania FPGA

5. Ignorujemy ostrzeżenia o braku modułów *debug core* w projekcie – istotnie, nie był on przez nas wykorzystywany: *INFO: [Labtools 27-1434] Device xc7a35t (JTAG device index = 0) is programmed with a design that has no supported debug core(s) in it. WARNING: [Labtools 27-3361] The debug hub core was not detected.*
6. Nasz projekt powinien w tym momencie już być uruchomiony w układzie FPGA. Widać to np. po zmianie sposobu migania diodami LED na płycie ewaluacyjnej.

4.3.5 Uruchomienie algorytmu przetwarzania

Na komputerze uruchamiamy środowisko Matlab za pomocą odpowiedniej ikony umieszczonej na pulpicie. Zmieniamy katalog roboczy na `c:\rim\fpga` i w wektorze `x` generujemy dane wejściowe dla algorytmu przetwarzania:

```
cd c:\rim\fpga
x=30000*(sin((0:1023)*2*pi/100));
```

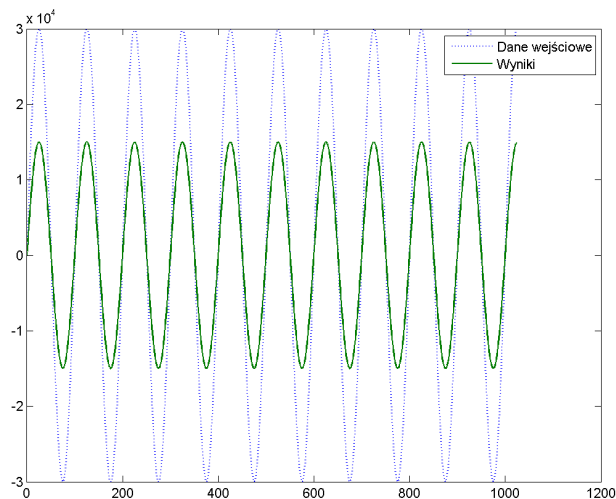
Następnie uruchamiamy algorytm przetwarzania i odczytujemy wynik jego działania do wektora `y`:

```
>> y=fpga(x);
Algorytm przetwarzania w FPGA zakończył pracę.
Czas przetwarzania danych : 0.04096 ms
Liczba wpisanych słów wynikowych : 1024
Liczba cykli zegara na przetwarzanie : 2048
Liczba cykli zegara na słowo wyniku : 2
Odczytano dane wynikowe z układu FPGA.
```

Po zakończeniu pracy algorytmu i odczytania wyników możemy przedstawić je na wspólnym wykresie razem z danymi wejściowymi:

```
t=0:1023;
plot(t,x,'--k',t,y,'-r'); grid;
legend('Dane wejściowe','Wyniki')
```

Na rys. 4.12 przedstawiono przykładowy wykres uzyskany w ten sposób po uruchomieniu algorytmu z wyłącznikiem SW0 w pozycji '0' (podział liczby przez 2).



Rysunek 4.12: Przykładowy wykres danych z algorytmu uruchomionego w układzie FPGA

4.4 Badanie algorytmu filtracji FIR

W ramach ćwiczenia będzie implementowany i badany algorytm filtracji FIR (patrz p. 2.1) rzędu $N = 8$. Algorytm pracuje w arytmetyce liczb stałoprecyzyjnych ze znakiem, typowej

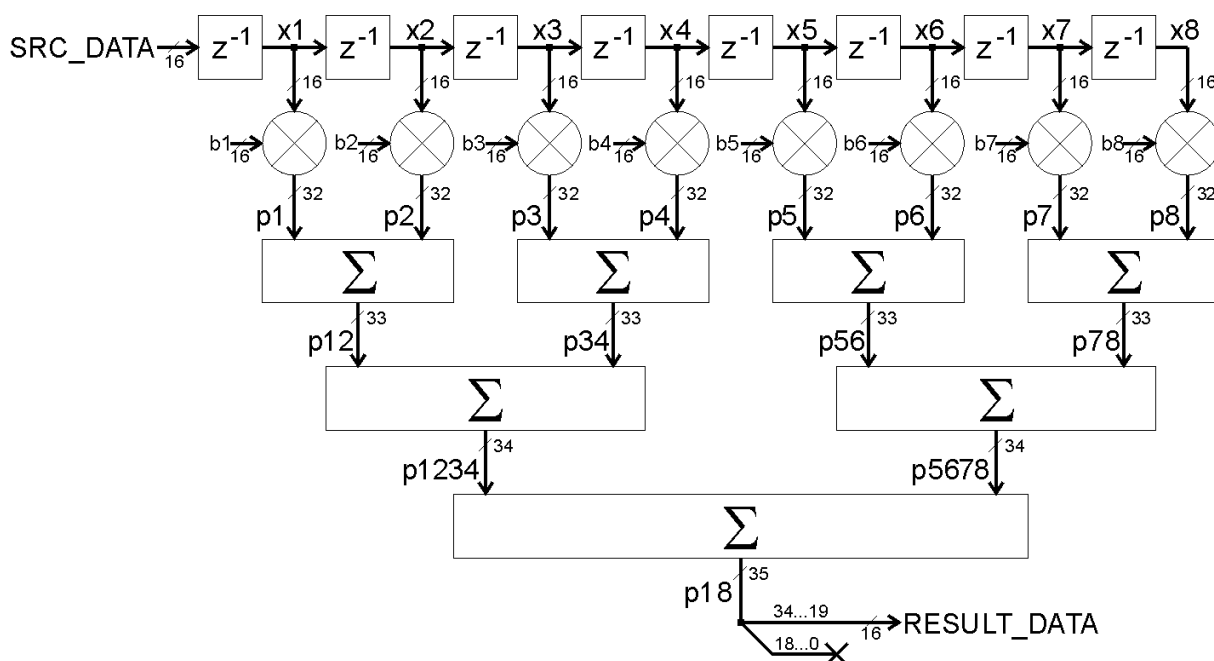
dla technologii FPGA. Wektor współczynników filtru ma postać:

$$\mathbf{b} = (16384, 8192, 4096, 2048, 1024, 512, 256, 128) \quad (4.1)$$

Dane wejściowe do filtracji będą pobierane z portu SRC_DATA komponentu rimlab01, zaś wyniki będą wpisywane do portu RESULT_DATA.

4.4.1 Implementacja filtru FIR w języku VHDL

Ten punkt należy zrealizować w domu podczas przygotowywania się do laboratorium. Bazując na przykładowym programie `fpgatop.vhd` (patz p. 4.3.2) należy stworzyć plik `fir1.vhd` (z jednostką projektową o nazwie `fir1`) realizujący operację filtracji pokazaną na rys. 4.13. Plik ten powinien być wybrany jako **główny plik projektu** (opcja *Set as Top Module* z menu kontekstowego).



Rysunek 4.13: Schemat blokowy filtru FIR

Jedną z zalet języków opisu sprzętu jest możliwość łatwego i czytelnego opisu nawet złożonych projektów za pomocą kilku linii kodu, o czym przekonamy się podczas wykonywania ćwiczenia. Podobne do siebie fragmenty kodu można w wygodny sposób kopiować i wykorzystywać wielokrotnie wewnątrz projektu.

Podczas implementacji filtru z rys. 4.13 w języku VHDL należy mieć na uwadze następujące wskazówki:

- Współczynniki filtru $b_1 \dots b_8$ należy zadeklarować jako stałe o wartościach podanych w (4.1), np.:

```
constant b1: STD_LOGIC_VECTOR (15 downto 0) := X"4000";
```

- Przy mnożeniu dwóch liczb K -bitowych podwaja się szerokość słowa danych. Należy to uwzględnić przy deklaracji sygnałów przechowujących wyniki mnożenia.

- Przy dodawaniu dwóch liczb, szerokość słowa zwiększa się o 1 bit. Należy to uwzględnić przy deklaracji sygnałów $p1...p8$ z kolejnymi wynikami sumowania, np:

```
signal p12 : STD_LOGIC_VECTOR (32 downto 0);
```

- Poszerzenie słowa należy uwzględnić również w samej operacji dodawania (koniecznie należy wskazać właściwą kolejność działań za pomocą nawiasów):

```
p12 <= (p1(31) & p1) + (p2(31) & p2);
```

- Operacje: pobierania próbek wejściowych SRC_DATA z portu DOUT_DATA komponentu rimlab01, przesuwania historii próbek oraz obliczania sum należy wykonywać synchronicznie, np. w stanie RD_DAT procesu fill_mem, odpowiednio aktualizując zawartość rejestrów $x1...x8$.
- Z ostatniej sumy należy wybrać najstarsze 16 bitów (tj. $p18(34 \text{ downto } 19)$) jako wynik RESULT_DATA wpisywany do portu DIN_RESULT komponentu rimlab01. Jest to skrajnie ostrożne podejście, które zagwarantuje brak błędów przepełnienia wyniku, niezależnie od warunków pracy algorytmu.

Operację mnożenia będziemy wykonywać, instalując w głównym pliku naszego projektu odpowiednią ilość razy „uniwersalny moduł mnożący” (komponent mult16x16):

```
mult1: mult16x16 port map (clk => clk, a => x1, b => b1, p => p1);
mult2: mult16x16 port map (clk => clk, a => x2, b => b2, p => p2);
-- itd.
```

Uwaga! Instancje modułów mult1...mult8 mnożących mult16x16, podobnie jak wszystkich innych komponentów, należy **umieszczać poza procesem fill_mem**, np. po instancji bloku komunikacyjnego rim1 : rimlab01 PORT MAP(...);.

Plik źródłowy mult16x16.vhd musi być dodany do struktury projektu (menu: *Sources / Add Sources*), zgodnie z opisem w p. 4.3.1. W przeciwnym wypadku nie będzie możliwe skompilowanie projektu.

Jeżeli w projekcie obok pliku źródłowego fir1.vhd znajduje się także poprzedni plik nadrzędny (fpgatop.vhd), to należy go usunąć, klikając prawym przyciskiem myszy i wybierając opcję *Remove* z menu kontekstowego.

4.4.1.1 Moduł mnożący mult16x16

W pliku mult16x16.vhd zapisany jest kod źródłowy uniwersalnego modułu mnożącego. Moduł ten, w zależności od parametrów podanych w klauzuli generic, może wykonywać mnożenie liczb całkowitych ze znakiem albo bez znaku, z zadaną szerokością słowa. Sama operacja mnożenia jest implementowana automatycznie w procesie syntezy logicznej. W zależności od parametru impl_dsp48, mnożenie może być wykonywane w dedykowanych blokach obliczeniowych (DSP) układu FPGA, albo w komórkach logicznych (Slice / LUT).

Definicja komponentu mnożącego, którą należy umieścić w głównym pliku projektowym, ma postać:

```
component mult16x16 is
  generic (
    word_size : natural := 16;
```



```

        signed_mult : boolean    := true;
        impl_dsp48   : string     := "yes"
    );
    port (
        clk : in    std_logic;
        a   : in    std_logic_vector(1*word_size-1 downto 0);
        b   : in    std_logic_vector(1*word_size-1 downto 0);
        p   : out   std_logic_vector(2*word_size-1 downto 0)
    );
end component;

```

Komponent jest opisany za pomocą trzech parametrów `generic`:

- `word_size` – szerokość słowa danych wejściowych,
- `signed_mult` – czy mnożenie jest wykonywane na liczbach ze znakiem,
- `impl_dsp48` – sposób implementacji; istotne dla nas są dwie możliwości:
 - "yes" – wykorzystanie dedykowanych bloków obliczeniowych DSP,
 - "no" – wykorzystanie bloków logicznych ogólnego przeznaczenia (Slice / LUT).

Obok każdego parametru podane są jego domyślne wartości, przyjmowane w przypadku umieszczenia w projekcie komponentu bez ustawiania parametrów (`generic map`).

4.4.1.2 Uruchomienie i badanie algorytmu

Po bezbłędnym skompilowaniu projektu można podjąć się uruchomienia go na sprzęcie. Układ FPGA należy zaprogramować konfiguracją z pliku `fir1.bit`) zgodnie z opisem w p. 4.3.4 – zaczynając z okna *Flow Navigator: Program and Debug / Open Hardware / Open Target / Program Device*.

W środowisku Matlab należy odpowiednio zmienić katalog bieżący oraz wygenerować ciąg danych wejściowych, zawierający powtarzane okresowo impulsy:

```

cd c:\rim\fpga
x=zeros(1,1024);
x(1:16:1008)=32767;

```

Pierwszy impuls można zastąpić sygnałem schodkowym:

```

x(1:20)=32767;

```

Po wygenerowaniu sygnału pobudzenia w wektorze `x` należy uruchomić algorytm w układzie FPGA oraz wykreślić wyniki obliczeń wraz z danymi wejściowymi:

```

y=fpga(x);
t=0:1023;
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 64 0 33000]); grid;

```

Na wykresie będzie widać: na czarno – impulsy pobudzające, oraz na czerwono – odpowiedź filtru.

Należy odpowiedzieć na następujące pytania:

1. Jakie jest przesunięcie wyniku (w cyklach zegara) od danych wejściowych i dlaczego?
2. Jaka jest amplituda odpowiedzi na pobudzenie impulsowe i schodkowe?
3. Jaka może być maksymalna teoretyczna amplituda sygnału, przy 16-bitowej szerokości słowa ze znakiem?
4. Ile cykli zegara trwa obliczanie każdego słowa wyniku?
5. Ile mnożeń i dodawań wymaga obliczanie każdego słowa wyniku obliczeń?
6. Ile zasobów logicznych (LUT, FF) oraz bloków obliczeniowych (DSP) zajęto?

4.4.1.3 Optymalizacja dynamiki

Za pomocą środowiska Matlab należy obliczyć teoretyczną amplitudę odpowiedzi filtru na dwa wykorzystywane do tej pory rodzaje pobudzeń:

1. pobudzenie impulsem o amplitudzie 32767,
2. pobudzenie schodkiem o wysokości 32767.

W wykonywaniu tych obliczeń można skorzystać np. z funkcji `conv()` obliczającej splot oraz zdefiniowanego już wektora danych wejściowych `x`).

Analizując wyniki powyższych obliczeń należy wyznaczyć optymalne przesunięcie bitowe wyniku, które pozwoli najlepiej wykorzystać 16-bitową dynamikę na wyjściu filtru, wspomagając się funkcją `log2()` środowiska Matlab. Na tej podstawie należy zmienić w kodzie źródłowym `fir1.vhd` wybór wyniku końcowego, kierowanego do sygnału `RESULT_DATA` z ostatniej obliczanej sumy: (p18).

Przed modyfikacją kodu i uruchomieniem kolejnej kompilacji należy najpierw zamknąć okno *Hardware Manager* środowiska Vivado. Po skompilowaniu projektu należy ponownie załadować konfigurację do układu FPGA (patrz p. 4.3.4) oraz sprawdzić w środowisku Matlab działanie zmodyfikowanego algorytmu:

```
y=fpga(x);
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki'); grid;
```

1. Jaka jest tym razem amplituda odpowiedzi na pobudzenie impulsowe i schodkowe?
2. Czy otrzymana wartość jest zgodna z obliczeniami teoretycznymi?
3. W jakim stopniu wykorzystano 16-bitową dynamikę filtru?
4. Czy w algorytmie może dojść do błędu przepełnienia (przesterowania)?

4.4.1.4 Optymalizacja szybkości przetwarzania

W pliku źródłowym `fir1.vhd` należy wprowadzić zmiany, tak aby w każdym cyklu zegara było jednocześnie pobierane jedno słowo danych z portu `DOUT_DATA`) i wpisywane jedno słowo wyniku do portu `DIN_RESULT`). Operacje te mogą być wykonywane w jednym stanie, np. `RD_DAT`, bez konieczności przechodzenia do stanu `WR_DAT`.

Przed modyfikacją kodu i uruchomieniem kompilacji należy zamknąć okno *Hardware Manager*. Zmodyfikowany kod źródłowy należy skompilować (patrz p. 4.3.3), a następnie przesłać konfigurację do układu FPGA (p. 4.3.4). Ponownie należy sprawdzić poprawność działania algorytmu w środowisku Matlab:

```
y=fpga(x);
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 64 0 33000]);
```

Jeśli na wykresie uzyskano prawidłowe przebiegi danych wynikowych, należy odpowiedzieć na pytania:

1. Ile cykli zegara trwa obliczanie każdego słowa wyniku?
2. Ile mnożeń i dodawań jest realizowane równolegle w każdym cyklu zegara?
3. Czy jest możliwe dalsze przyspieszenie (zrównoleglenie) obliczeń? Jakich zmian w architekturze by to wymagało? Czy układ FPGA udostępnia niezbędne do tego zasoby?
4. Ile zasobów logicznych (LUT, FF) oraz bloków obliczeniowych (DSP) zajęto?
5. Ile filtrów tego rodzaju „zmieści się” w układzie FPGA? Z czego wynika ograniczenie?

4.4.2 Implementacja mnożenia w komórkach logicznych

Do tej pory operacja mnożenia była implementowana z wykorzystaniem dedykowanych bloków obliczeniowych, gdyż takie preferencje wskazaliśmy parametrem `impl_dsp48`. To w niektórych przypadkach może okazać się opłacalne, lub nawet konieczne może być realizowanie mnożenia w komórkach logicznych ogólnego przeznaczenia (Slice / LUT). Taka realizacja była jedyną możliwością w starszych rodzinach układów FPGA, nie wyposażonych jeszcze w bloki DSP ani nawet w bloki mnożące.

W tym punkcie sprawdzimy możliwość realizacji mnożenia w blokach Slice / LUT, bez zajmowania dedykowanych bloków obliczeniowych DSP48. W kodzie źródłowym projektu, w definicji modułu `mult16x16` należy wyłączyć wykorzystanie bloków DSP48, zmieniając wartość parametru `impl_dsp48` na "no":

```
component mult16x16 is
  generic (
    word_size      : natural      := 16;
    signed_mult    : boolean      := true;
    impl_dsp48     : string       := "no");
```

Przed modyfikacją kodu i uruchomieniem kompilacji należy zamknąć okno *Hardware Manager*. Kod źródłowy po wprowadzeniu zmian należy skompilować (p. 4.3.3), a następnie uruchomić w układzie FPGA (p. 4.3.4). W środowisku Matlab należy sprawdzić poprawność działania algorytmu:

```
y=fpga(x);
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 64 0 33000]);
```

Jeśli uzyskane wykresy są zgodne z przewidywaniami, należy odpowiedzieć na pytania:

1. Ile zasobów logicznych (LUT, FF) oraz bloków obliczeniowych (DSP) zajęto?
2. Jak wygląda zajętość zasobów w porównaniu z poprzednią realizacją, wykorzystującą bloki mnożące?
3. Ile czasu trwała implementacja projektu (*Design Runs / impl_1 / Elapsed*)?
4. Jaki jest koszt realizacji mnożenia i z czego on wynika? (Porównaj zajętości zasobów)
5. Ile filtrów tego rodzaju „zmieści się” w układzie FPGA? Z czego wynika ograniczenie?

4.4.3 Implementacja filtru o zmiennych współczynnikach

W tej wersji zastosujemy zmienne współczynniki filtru, przechowywane w rejestrach *c1...c8*:

```
signal c1,c2,c3,c4,c5,c6,c7,c8 : STD_LOGIC_VECTOR (15 downto 0);
```

Sygnały *c1...c8* należy dołączyć do odpowiednich modułów mnożących zamiast stałych *b1...b8*. Współczynniki te będą inicjowane początkowymi 8 słowami z pamięci danych, odczytanymi z portu *SRC_DATA*. Operację programowania współczynników najłatwiej będzie zaimplementować, stosując przypisania warunkowe w stanie *RD_DAT*:

```
elsif stan_alg=RD_DAT then -- Odczytanie słowa z pamięci danych
    if cnt_data=0 then
        c1 <= SRC_DATA;
    end if;
    if cnt_data=1 then
        c2 <= SRC_DATA;
    end if;
    -- itd.
```

Uwaga! Sygnał zezwolenia odczytu *SRC_RD* musi być ustawiony jako '1' w chwili przejścia do stanu *RD_DAT*, czyli równolegle z przypisaniem: *stan_alg<=RD_DAT*. Jeśli sygnał będzie ustawiony później, to pierwsza komórka pamięci zostanie odczytana dwukrotnie (do obu rejestrów: *c1, c2*) i odpowiedź impulsowa filtru zostanie przesunięta o jeden współczynnik.

Projekt po tych modyfikacjach zmian należy skompilować i uruchomić w układzie FPGA. W środowisku Matlab należy wygenerować nowe dane wejściowe, uwzględniające na początku programowanie współczynników filtru:

```
x=[16384 8192 4096 2048 1024 512 256 128 zeros(1,1016)];
```

W danych wejściowych umieścimy również sygnał schodkowy oraz okresowo powtarzane impulsy:

```
x([16:32 48:16:1008])=32767;
x(48:16:1024)=32767;
```

Po przygotowaniu danych, uruchamiamy algorytm na układzie FPGA i sprawdzamy poprawność jego działania na wykresie:

```
y=fpga(x);
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 80 0 33000]); grid;
```

Możemy też sprawdzić, czy mechanizm programowania współczynników działa prawidłowo, i czy operacje arytmetyczne są prawidłowo wykonywane na liczbach ujemnych. W tym celu zmieniamy drugą połowę współczynników filtra na wartości ujemne, tworząc tym samym rodzaj filtra różniczkującego:

```
x(5:8)=[ -2048 -4096 -8192 -16384];
y=fpga(x);
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 80 -33000 33000]); grid;
```

Jeśli uzyskane wykresy są zgodne z przewidywaniami, należy odpowiedzieć na pytania:

1. Ile zasobów logicznych (LUT, FF) oraz bloków obliczeniowych (DSP) zajęto?
2. Jak wygląda zajętość zasobów w porównaniu z poprzednią realizacją, opartą na stałych współczynnikach?
3. Ile czasu trwała implementacja projektu (*Design Runs / impl_1 / Elapsed*)?
4. Jaki jest koszt realizacji operacji mnożenia w tym przypadku i z czego on wynika?
5. Ile filtrów tego rodzaju „zmieści się” w układzie FPGA? Z czego wynika ograniczenie?

4.4.4 Filtr o zm. współczynnikach, implementacja w blokach DSP

Przełączmy implementację mnożenia znów na wykorzystanie dedykowanych bloków obliczeniowych DSP48:

```
component mult16x16 is
  generic ( word_size : natural := 16; signed_mult : boolean
           := true;
           impl_dsp48 : string := "yes"
         );
```

Kompilujemy (p. 4.3.3) i uruchamiamy w układzie FPGA (p. 4.3.4) projekt po tej modyfikacji. W środowisku Matlab sprawdzamy poprawność działania algorytmu:

```
y=fpga(x);
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 80 0 33000]); grid;
```

Jeśli uzyskane wykresy są zgodne z przewidywaniami, należy odpowiedzieć na pytania:

1. Ile zasobów logicznych (LUT, FF) oraz bloków obliczeniowych (DSP) zajęto?
2. Jak wygląda zajętość zasobów w porównaniu z poprzednią realizacją na kom. LUT?
3. Ile czasu trwała implementacja projektu (*Design Runs / impl_1 / Elapsed*)?

4. Jaki jest koszt realizacji operacji mnożenia w tym przypadku i z czego on wynika?
5. Kiedy warto stosować dedykowane bloki DSP, a kiedy komórki logiczne Slice / LUT?

Rozdział 5

Biblioteki obliczeniowe dla układów FPGA

Celem ćwiczenia jest zapoznanie się z możliwościami implementacji złożonych funkcji obliczeniowych w układzie FPGA z wykorzystaniem modułów dostarczanych w bibliotece *IP Catalog* środowiska projektowego *Vivado*.

W ramach ćwiczenia badane będą różne sposoby implementacji algorytmu filtracji FIR, szybkiej dyskretnej transformaty Fouriera (FFT) oraz szybkiego algorytmu aproksymacji modułu liczby zespolonej.

Podczas laboratorium należy wypełniać formatkę sprawozdania, odpowiadając krótko i konkretnie na postawione pytania. Należy zgłaszać prowadzącemu każde wykonane zadanie i możliwie szybko zgłaszać ew. trudności w jego wykonaniu. Przed laboratorium należy wykonać w domu zadania: [5.4.1](#), [5.4.2](#).

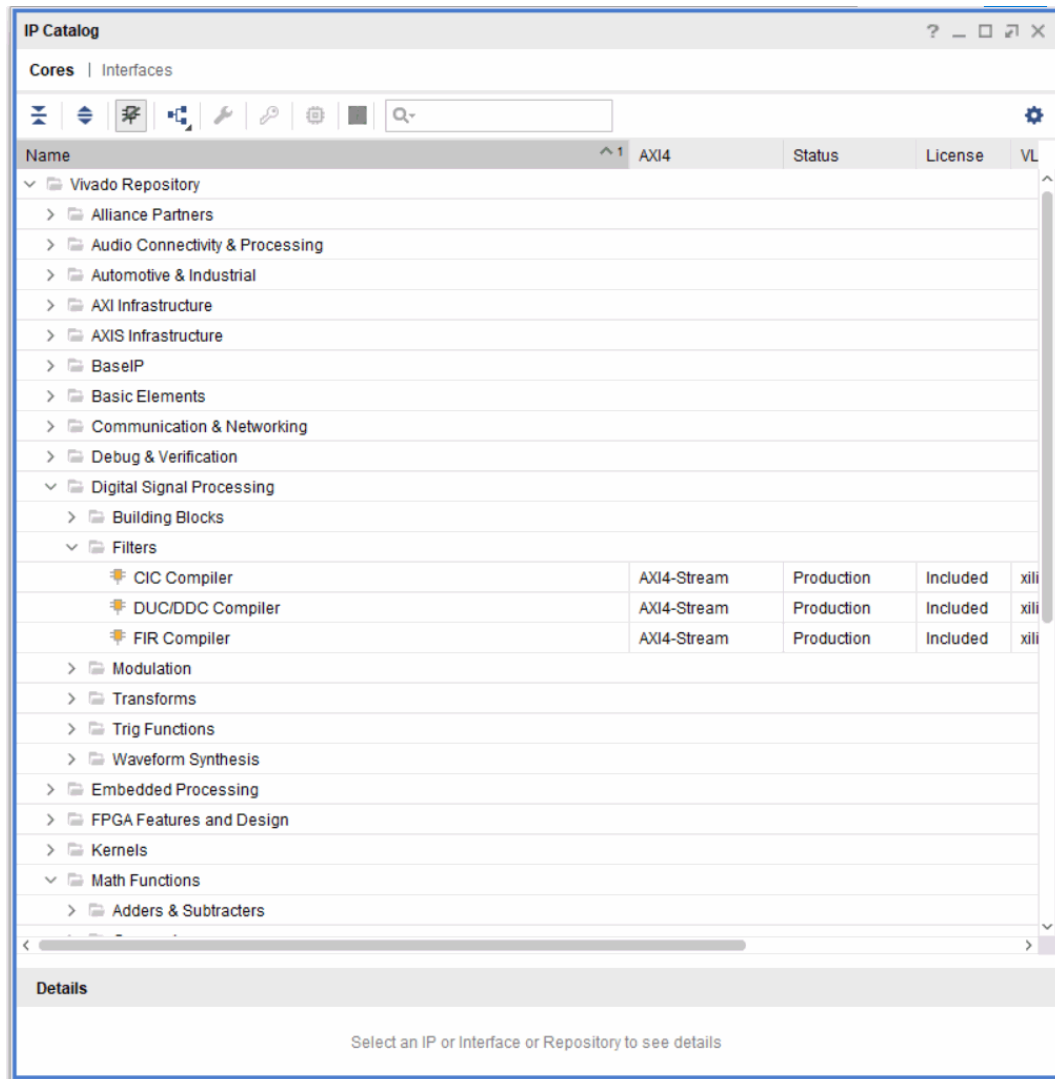
5.1 Biblioteka IP Catalog

Biblioteka *IP Catalog* jest częścią środowiska projektowego *AMD/Xilinx Vivado*. Umożliwia ona generowanie specjalizowanych komponentów IP (bloków bibliotecznych *Intellectual Property* – tzw. *IP Core*) realizujących złożone zadania, jak np:

- funkcje arytmetyczne (np. dzielenie, pierwiastkowanie, mnożenie liczb zespolonych),
- obliczenia funkcji trygonometrycznych i hiperbolicznych,
- transformaty FFT i DFT,
- obliczenia na liczbach zmiennoprzecinkowych,
- algorytmy cyfrowego przetwarzania sygnałów (DSP),
- kontrolery pamięci (w tym np. DDR2, DDR3),
- interfejsy wejścia / wyjścia (np. PCI Express),
- bloki sieciowe i telekomunikacyjne (np. warstwa MAC protokołu Ethernet).

Część komponentów z katalogu dostępna jest do wykorzystania w darmowej licencji środowiska *Vivado*, zaś pozostałe (zaznaczone na szaro w katalogu) wymagają wykupienia licencji.

Na ćwiczeniu biblioteka *IP Catalog* będzie wykorzystana do generowania komponentów realizujących w różnych architekturach filtrację FIR oraz transformatę FFT, opisanych dalej w instrukcji.



Rysunek 5.1: Fragment katalogu dostępnych modułów IP

5.2 Synteza filtru FIR (moduł FIR Compiler)

Kreator *FIR Compiler* jest jednym z elementów biblioteki *IP Catalog* i jest zlokalizowany w folderze Digital Signal Processing / Filters (patrz rys.5.1). Parametry tego modułu zestawiono w tab. 5.1. Umożliwia on wygenerowanie gotowego komponentu realizującego jeden z wybranych algorytmów filtracji cyfrowej:

- klasyczny filtr o skończonej odpowiedzi impulsowej (FIR),

- półpasnowy filtr FIR,
- transformator Hilberta,
- interpolowany filtr FIR,
- polifazowy decymator lub interpolator,
- półpasnowy decymator lub interpolator,
- polifazowy bank filtrów.

Większość generowanych filtrów ma opcję pracy wielokanałowej. Możliwe jest także zrealizowanie banku filtrów po wprowadzeniu kilku różnych zestawów współczynników.

Dość istotnym ograniczeniem kreatora *FIR Compiler* jest możliwość syntezy wyłącznie filtrów z rzeczywistymi współczynnikami. W przypadku realizacji filtru o współczynnikach zespolonych, konieczne jest ręczne opisanie jego struktury w kodzie VHDL, podobnie jak w poprzednim ćwiczeniu, albo odpowiednie zrównoleglenie i połączenie trzech filtrów o współczynnikach rzeczywistych.

Tabela 5.1: Możliwości syntezy filtrów w module *FIR Compiler*

Właściwość	Struktura bezpośrednia	Struktura transponowana
Liczba współczynników (N)	2...1024	2...1024
Szerokość słowa współczynników	2...35	2...35
Szerokość słowa danych B	2...35	2...35
Liczba kanałów	1...64	1
Liczba zestawów współczynników	1...256	1...256
Zaokrąglanie wyniku	TAK	TAK

W ramach ćwiczenia będzie wykorzystywany wyłącznie pierwsza, najbardziej ogólna odmiana filtru, tj. klasyczny filtr FIR o zadanych współczynnikach. W filtrze tym realizowany jest splot sygnału wejściowego $x(n)$ z odpowiedzią impulsową filtru $a(n)$:

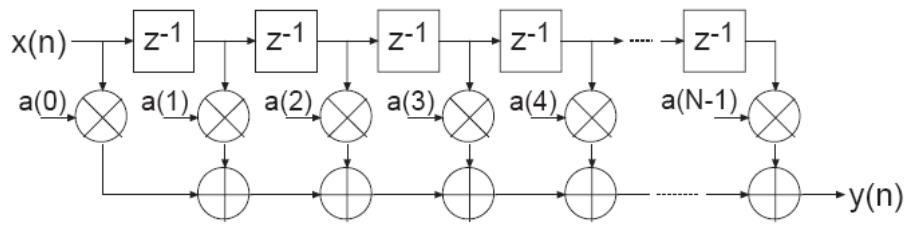
$$y(n) = x(n) * a(n). \quad (5.1)$$

Odpowiedź filtru $a(n)$ składa się z N stałych współczynników. Dla każdej próbki sygnału wyjściowego $y(n)$ obliczany jest zatem iloczyn skalarny:

$$y(n) = \sum_{k=0}^{N-1} a(k)x(n-k). \quad (5.2)$$

Graficznie przedstawiono to na rys. 5.2. Tradycyjnie blozki oznaczone jako z^{-1} symbolizują opóźnienie danych o jeden cykl zegara, czyli elementarne komórki pamięci lub rejestry.

Szerokość słowa danych zwiększa się po każdej operacji mnożenia i dodawania. W rezultacie na wyjściu filtru otrzymywane jest znacznie szersze słowo $y(n)$ w pełnej precyzji (por. p. 4.4.1). Operacja liczenia splotu (5.2) może być zaimplementowana w jednej z trzech struktur:



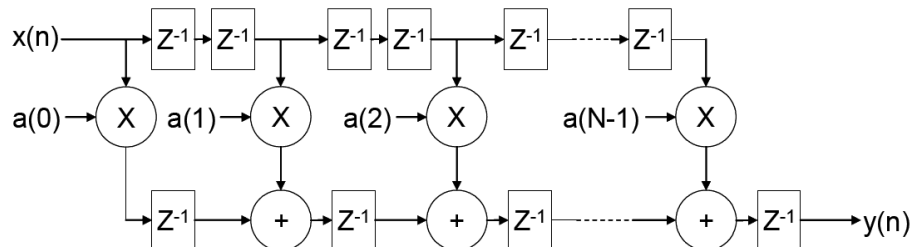
Rysunek 5.2: Schemat blokowy filtra FIR w strukturze bezpośredniej

1. w **strukturze bezpośredniej** z potokowaniem (*Systolic Multiply-Accumulate*),
2. w strukturze transponowanej (*Transpose Multiply-Accumulate*).

Obie struktury wykorzystują dedykowane bloki obliczeniowe DSP układu FPGA. Pierwsza struktura filtra będzie badana podczas ćwiczenia.

5.2.1 Implementacja filtra FIR w strukturze bezpośredniej

W *strukturze bezpośredniej* z potokowaniem, pokazanej na rys. 5.3, umieszczono dodatkowe rejestry na wyjściach symatorów (porównaj z rys. 5.2). W skutek tego w każdym cyklu zegara obliczana jest suma tylko dwóch składników (po pierwszym mnożeniu przez $a(0)$ nie ma jeszcze sumowania), a nie wszystkich N składników, tak jak w modelu matematycznym (5.2). Dzięki temu w układzie implementowane są znacznie prostsze sumatory (tzn. krótsze łańcuchy logiki kombinacyjnej) i filtr jako całość może pracować ze znacznie większą częstotliwością zegara, osiągając przy tym większą przepustowość. Aby wyrównać opóźnienia w obu ścieżkach danych, konieczne jest również wstawienie dodatkowych rejestrów opóźniających sygnał $x(n)$ w górnej ścieżce.

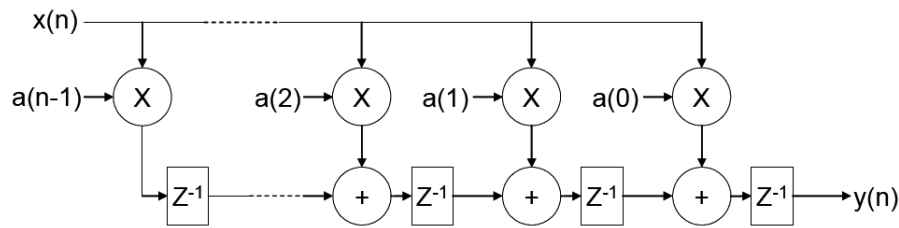


Rysunek 5.3: Bezpośrednia struktura filtra FIR z potokowaniem

5.2.2 Implementacja filtra FIR w strukturze transponowanej

Strukturę transponowaną filtra, pokazaną na rys. 5.4, uzyskuje się ze struktury bezpośredniej (pokazanej na rys. 5.2) w trzech krokach:

1. zamiana wejścia z wyjściem,
2. odwrócenie kierunku przesyłania sygnału,
3. zastąpienie sumatorów rozgałęzieniami, i na odwrót.

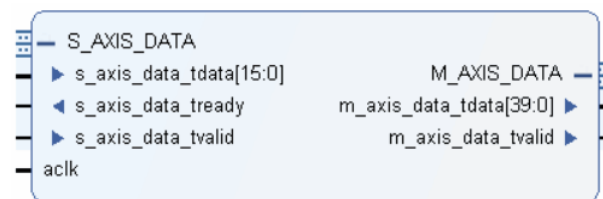


Rysunek 5.4: Transponowana struktura filtru FIR

Zaletami struktury transponowanej jest mniejsza liczba wykorzystanych przerzutników¹ oraz skrócenie opóźnienia potokowania o jeden cykl zegara.

5.2.3 Wyprowadzenia modułu FIR

Na rys. 5.5 pokazano symbol blokowy filtru FIR, generowanego w kreatorze *FIR Compiler*. Część portów jest opcjonalna i generowana tylko w szczególnych konfiguracjach.



Rysunek 5.5: Wejścia i wyjścia modułu filtracji FIR

Moduł filtru FIR, tak jak prawie wszystkie komponenty dostępne w bibliotece *IP Catalog*, do przesyłania danych na wejściu i wyjściu wykorzystuje standardowe magistrale *AXI4-Stream*: *S_AXIS_DATA*, *M_AXIS_DATA*. Jest to bardzo dobrze opisana i szeroko przyjęta magistrala przeznaczona do szybkiego i łatwego przesyłania strumieni danych, tzn. takich, w których słowa nie wymagają adresowania pamięci. Szczegółowe omówienie tego standardu wykracza poza zakres ćwiczenia, ograniczymy się do wyjaśnienia tylko niezbędnych portów:

- *X_axis_data_tdata* – szyna danych (o szerokości $K \times 8$ bitów,
- *X_axis_data_tready* – flaga sygnalizująca gotowość do odbioru słowa danych,
- *X_axis_data_tvalid* – sygnalizacja poprawności słowa danych na szynie *_tdata*,
- *aclk* – sygnał zegarowy.

Znak „x” na początku nazwy wszystkich portów magistrali jest zastąpiony znakiem „s” w przypadku interfejsu *Slave* albo „m” w przypadku interfejsu *Master*. Interfejs *Slave* odbiera dane, tzn. odczytuje / monitoruje stan portów wejściowych (*tdata*, *tvalid*) i steruje flagą *tready*. Interfejs *Master* odwrotnie – wysyła dane: steruje portami *tdata*, *tvalid* i opcjonalnie kontroluje stan portu wejściowego *tready*.

¹We współczesnych układach FPGA, bogatych w przerzutniki i z opcją realizacji opóźnień w tablicach LUT, ten argument nie jest już tak istotny.

Każde połączenie magistrali *AXI4-Stream* musi być typu *Master – Slave, Point-to-Point*, czyli bez rozgałęzień, i polega na połączeniu ze sobą portów o jednakowych nazwach. Transfer słowa danych następuje przy narastającym zboczu zegara `aclk` pod warunkiem, że obie flagi kontroli przepływu (`trready`, `tvalid`) są jednocześnie w stanie aktywnym (wysokim). Sygnał zegarowy `aclk` w przypadku omawianego modułu filtru FIR jest wspólny dla obu interfejsów (*Master, Slave*), jednak w ogólnie nie jest to regułą.

5.3 Implementacja alg. FFT (moduł *Fast Fourier Transform*)

Algorytm FFT² jest szybką implementacją obliczenia dyskretnej transformaty Fouriera (DFT) określonej wzorem:

$$X(k) = \sum_{n=0}^{N-1} x(n) \exp(-j2\pi nk/N), \quad (5.3)$$

gdzie N jest rozmiarem transformaty DFT, tzn. liczbą próbek sygnału na wejściu i liczbą próbek widma na wyjściu transformaty. Algorytm FFT znajduje zastosowanie dla rozmiaru transformaty będącego potęgą dwójki ($N = 2^k$) i pozwala zmniejszyć liczbę mnożeń zespolonych z N^2 do około $\frac{N}{2} \log_2 N$. Dla dużych wartości N pozwala on zatem znacznie zmniejszyć nakłady obliczeniowe. Warto podkreślić, że algorytm FFT oblicza przy tym dokładnie taki sam wynik, jak dyskretna transformata Fouriera (nie jest aproksymacją DFT!).

Implementacja algorytmu FFT jest zadaniem niezwykle pracochłonnym i skomplikowanym, szczególnie w układach logicznych. Zadanie to zostało jednak wykonane dla nas już dawno przez twórców środowiska projektowego dla układów FPGA. W rezultacie dostępny jest gotowy moduł biblioteczny *Fast Fourier Transform*³, zlokalizowany w *IP Catalog* w folderze: Digital Signal Processing / Transforms / FFTs (patrz rys. 5.1). Parametry bloku zestawiono w tab. 5.2.

Tabela 5.2: Parametry i możliwości modułu *Fast Fourier Transform*

Parametr	Wartość
Obliczanie prostej i odwrotnej transformaty Fouriera	TAK (przełączane podczas pracy)
Rozmiar transformaty N	$N = 2^k$, $k = 3 \dots 16$ (z opcją zmiany podczas pracy)
Szerokość słowa danych B	8...34
Dostępne tryby obliczeń	Stałoprzecinkowy z pełną precyzją, stałoprzecinkowy ze skalowaniem, zmiennoprzecinkowy
Sposób zaokrąglania wyników pośrednich	Obcinanie bitów lub zaokrąglanie
Reprezentacja danych	Stałoprzecinkowa lub zmiennoprzecinkowa

²Lyons R. *Wprowadzenie do cyfrowego przetwarzania sygnałów*, Rozdział 4, WKŁ 2006

³Xilinx, *Fast Fourier Transform v9.1 LogiCORE IP Product Guide*, https://www.xilinx.com/support/documents/ip_documentation/xfft/v9_1/pg109-xfft.pdf

Z wykorzystaniem modułu bibliotecznego *Fast Fourier Transform* możliwa jest implementacja algorytmu FFT w jednej z czterech architektur:

1. potokowej – strumieniowej,
2. o podstawie 4 (*Radix-4*),
3. o podstawie 2 (*Radix-2*),
4. uproszczonej o podstawie 2 (*Radix-2 Lite*).

W ramach ćwiczenia będą badane i porównywane implementacje FFT o podstawie 2 (*Radix-2*) i o podstawie 4 (*Radix-4*).

5.3.1 Architektura strumieniowa FFT

Architektura potokowa – strumieniowa zawiera szereg zrównoleglonych „motylków” (*Radix-2 Butterfly*, patrz rys. 5.7) o podstawie 2 wraz z własnymi blokami pamięci i zapewnia ciągłe przetwarzanie danych. W tym trybie pracy komponent jednocześnie wykonuje operacje na bieżącym bloku danych znajdującym się w pamięci, ładuje następny blok danych wejściowych, i wysyła na zewnątrz wyniki obliczeń z poprzedniego bloku.

Użytkownik może wprowadzić strumień danych wejściowych na wejście bloku, i po odczekaniu opóźnienia na wykonanie pierwszych obliczeń (napełnienie potoku), w sposób ciągły odbierać strumień danych wyjściowych z algorytmu FFT. Architektura strumieniowa, spośród wszystkich dostępnych, zajmuje największą ilość zasobów FPGA.

5.3.2 Architektura FFT z dekompozycją o podstawie 4

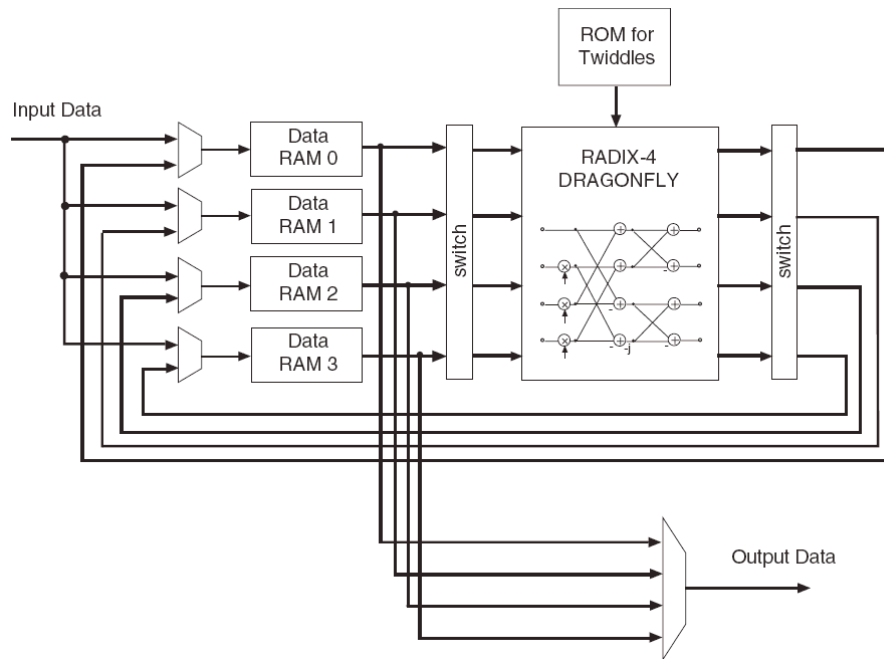
W architekturze FFT z dekompozycją o podstawie 4 (*Radix-4*), zilustrowanej na rys. 5.6, ładowanie bloku danych i odczytywanie wyników odbywa się oddzielnie od wykonywania obliczeń – operacje te nie są wykonywane jednocześnie. Po uruchomieniu algorytmu należy rozpocząć ładowanie danych wejściowych. Po skompletowaniu całego bloku N próbek, algorytm rozpoczyna obliczanie transformaty. Dopiero po zakończeniu obliczeń możliwe jest odczytanie bloku danych wynikowych. Operacje ładowania danych i odczytywania wyników mogą być wykonywane równolegle tylko wtedy, jeśli wyniki będą odczytywane z odwrotnym porządkiem bitów adresowych⁴.

Obliczenia są wykonywane w jednym jądrze o podstawie 4 (*Radix-4 Dragonfly*). Ten rodzaj architektury cechuje się mniejszą zajętością zasobów logicznych w porównaniu do architektury potokowo – strumieniowej, lecz jednocześnie wymaga dłuższego czasu na obliczenie transformaty.

5.3.3 Architektura FFT z dekompozycją o podstawie 2

W architekturze FFT z dekompozycją o podstawie 2 (*Radix-2*), zilustrowanej na rys. 5.7, ładowanie bloku danych i odczytywanie wyników tak jak poprzednio odbywa się oddzielnie od wykonywania obliczeń.

⁴Lyons R. *Wprowadzenie do cyfrowego przetwarzania sygnałów*, rozdz. 4.5



Rysunek 5.6: Architektura FFT z dekompozycją o podstawie 4

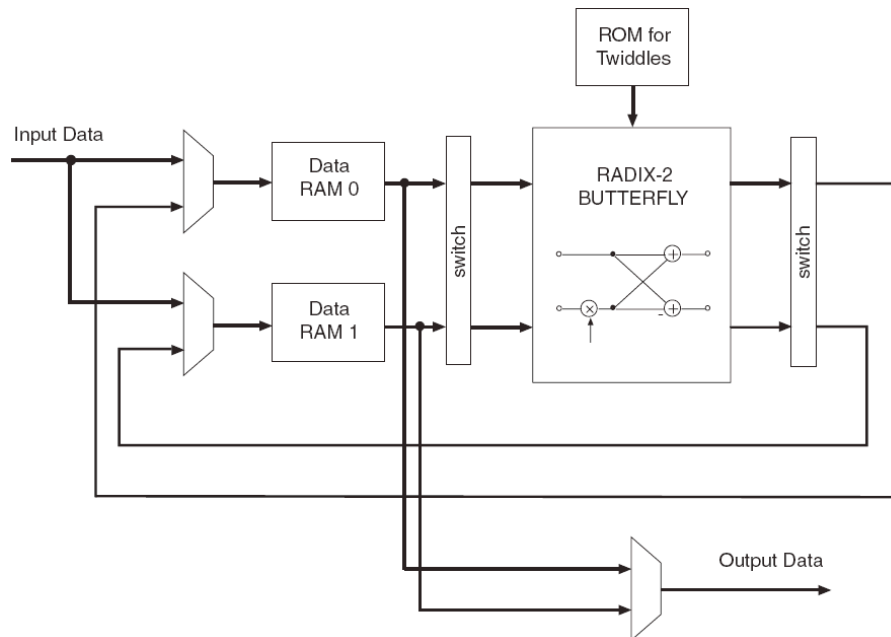
Obliczenia są wykonywane w jednym „motylku” o podstawie 2 (*Radix-2 Butterfly*). Proces przesyłania danych wejściowych i wyjściowych jest identyczny, jak w architekturze z dekompozycją o podstawie 4. W tym przypadku mamy jednak do czynienia z mniejszą zajętością zasobów logicznych i dłuższym czasem wykonywania obliczeń.

5.3.4 Uproszczona architektura FFT z dekompozycją o podstawie 2

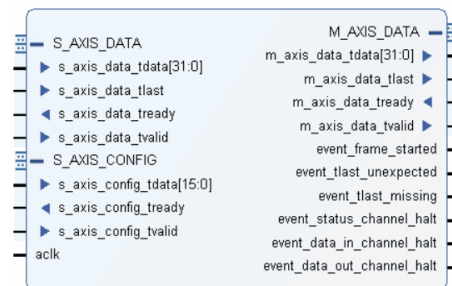
W uproszczonej architekturze FFT o podstawie 2 (*Radix-2 Lite*) „motylek”, w którym wykonywane są obliczenia, został zredukowany tylko jednego, współdzielonego akumulatora. W kolejnych cyklach zegara obliczane są na przemian części rzeczywiste i urojone transformaty. Skutkuje to zmniejszoną zajętością zasobów logicznych, lecz z drugiej strony – wydłużeniem czasu obliczeń. Sam proces przesyłania danych wejściowych i wyjściowych jest identyczny, jak w poprzednich architekturach.

5.3.5 Wprowadzenia bloku FFT

Na rys. 5.8 przedstawiono schematycznie wygląd komponentu *Fast Fourier Transform*.



Rysunek 5.7: Architektura FFT z dekompozycją o podstawie 2



Rysunek 5.8: Wejścia i wyjścia bloku transformaty FFT

5.4 Szybka aproksymacja modułu liczby zespolonej

Wynikiem działania transformaty FFT jest wektor N liczb zespolonych: $X(k) = X_{re}(k) + jX_{im}(k)$. Z naszego punktu widzenia istotny będzie tylko moduł danych wyjściowych – pomijamy informację o fazach poszczególnych składowych widma:

$$|X(k)| = \sqrt{X_{re}^2(k) + X_{im}^2(k)}. \quad (5.4)$$

Obliczenie modułu wymaga najpierw policzenia sumy kwadratów części rzeczywistej i urojonej (co jest zadaniem łatwym, jeśli mamy do dyspozycji dedykowane bloki mnożące), a następnie obliczenia pierwiastka z sumy.

Operacja pierwiastkowania może zostać zrealizowana np. w algorytmie CORDIC, jednak jest ona wykonywana iteracyjnie w wielu cyklach zegara. W wielu praktycznych przypadkach (w tym również podczas realizacji tego ćwiczenia) nie jest wymagany dokładny wynik pierwiastkowania i w zupełności wystarcza przybliżenie, ale za to operacja musi być wykonywana znacznie szybciej – najlepiej w jednym lub dwóch cyklach zegara.

Przybliżoną wartość modułu wektora $V = I + jQ$ można wyznaczyć, korzystając z algorytmu $\alpha\text{Max} + \beta\text{Min}$ ⁵ operującego na wartościach bezwzględnych części rzeczywistej i urojonej:

$$V_a = \alpha \cdot \max(|I|, |Q|) + \beta \cdot \min(|I|, |Q|) \approx |V|. \quad (5.5)$$

Istnieje kilka par wartości stałych α, β zapewniających różne dokładności aproksymacji modułu wektora. Kilka wybranych par stałych, zapewniających najmniejsze błędy aproksymacji, zestawiono w tab. 5.3. Wartości stałych zostały dobrane w ten sposób, aby możliwe było

Tabela 5.3: Wybrane wartości parametrów α, β

α	β	Największy błąd [dB]
1	1/4	-1,07
1	1/8	0,57
7/8	7/16	-1,16
15/16	15/32	-0,56

zrealizowanie operacji mnożenia i dzielenia za pomocą przesunięć bitowych i odejmowania, czyli z minimalną zajętością zasobów logicznych. Operację mnożenia przez 15/16 można zrealizować następująco: $\frac{15}{16}A = \frac{(A \ll 4) - A}{16} = [(A \ll 4) - A] \gg 4$.

5.4.1 Implementacja algorytmu w kodzie VHDL

W ramach przygotowania do ćwiczenia należy zakodować algorytm $\alpha\text{Max} + \beta\text{Min}$ z wybranym przez siebie z tab. 5.3 zestawem parametrów α, β jako samodzielny, syntezywalny moduł `mag16.vhd` w języku VHDL. Przyjęte kryterium wyboru parametrów należy uzasadnić prowadzącemu ćwiczenie. Do obliczeń arytmetycznych należy użyć biblioteki `ieee.numeric_std` udostępniającej obliczenia na typach danych `SIGNED`, `UNSIGNED` (w tym również funkcję wartości bezwzględnej `ABS`).

⁵Lyons R. *Wprowadzenie do cyfrowego przetwarzania sygnałów*, rozdz. 10.2

Przygotowany komponent `mag16.vhd` powinien pracować synchronicznie i potokowo, a jego porty wejścia i wyjścia powinny mieć następującą organizację:

```
entity mag16 is
    Port (
        in_re  : in  STD_LOGIC_VECTOR (15 downto 0);
        in_im  : in  STD_LOGIC_VECTOR (15 downto 0);
        clk    : in  STD_LOGIC;
        mag_out: out STD_LOGIC_VECTOR (15 downto 0)
    );
end mag16;
```

Przed przystąpieniem do ćwiczenia należy oszacować opóźnienie (liczone w cyklach zegara) wnoszone przez algorytm, wynikające z pracy potokowej.

5.4.2 Implementacja rejestru opóźniającego

Moduł `mag16.vhd` opóźnia dane wyjściowe o kilka cykli. Aby to opóźnienie skomensować, można równolegle opóźniać o taką samą liczbę cykli sygnał zezwalający na zapis danych wyników do pamięci (port `WR_RESULT` komponentu `rimlab00`).

Opóźnienie to powinno być zrealizowane w dodatkowym komponencie `delaysel.vhd`. Komponent ten należy przygotować w ramach przygotowania do ćwiczenia jako samodzielny, syntezywalny moduł w języku VHDL.

Komponent powinien pracować synchronicznie i opóźniać sygnał wejściowy o liczbę cykli `sel` zadaną jako parametr podczas kompilacji projektu. Deklaracja jednostki tego komponentu powinna wyglądać następująco:

```
entity delaysel is
    Generic (
        sel : integer := 8 -- domyślna wartość opóźnienia
    ); --
    Port ( din : in  STD_LOGIC;
          dout : out STD_LOGIC;
          clk  : in  STD_LOGIC
    );
end delaysel;
```

Wartość parametru `sel` ustawiono domyślnie na 8. Docelowo należy podać tutaj właściwą wartość opóźnienia, wynikającą z potokowej pracy bloku `mag16` (patrz p. 5.4.1).

5.5 Badanie bloku filtracji FIR

5.5.1 Utworzenie projektu

Na początku zajęć należy uruchomić system operacyjny *Windows* na komputerze dostępnym na stanowisku. Na pulpicie należy odszukać i uruchomić ikonę *Vivado 2018.2* (nie pomylić z *Vivado HLS 2018*).

Po uruchomieniu środowiska tworzymy nowy projekt za pomocą kreatora. W tym celu należy:

1. z okna *Quick Start* wybrać *Create Project*,
2. w nowym oknie *New Project* kliknąć na przycisku *Next*,
3. wpisać nazwę projektu *Project name*, np. `LAB5_Imię_Nazwisko` i ustawić jego lokalizację na dysku *Project location* w katalogu `c:\rim` – patrz rys. 4.3,
4. w oknie *Project Type* wybrać: *RTL Project* (domyślne ustawienie), kliknąć *Next*,
5. w oknie *Add Sources*: kliknąć *Add Files*, przejść do folderu: `c:\rim\fpga\rtl`,
6. zaznaczyć wszystkie widoczne pliki z folderu z wyjątkiem **mult16x16.vhd**, kliknąć na przycisku *OK* a następnie *Next*,
7. w oknie *Add Constraints* kliknąć *Add Files*, wskazać plik: `basys-rim.xdc`, kliknąć na przycisku *OK* a potem *Next*,
8. w oknie *Default Parts*, w zakładce *Parts* odnaleźć i z rodziny *Artix-7* wybrać układ: **xc7a35tcbg236-1** (patrz rys. 4.4), kliknąć na przycisku *Next*,
9. w oknie *New Project Summary* kliknąć na przycisku *Finish*.

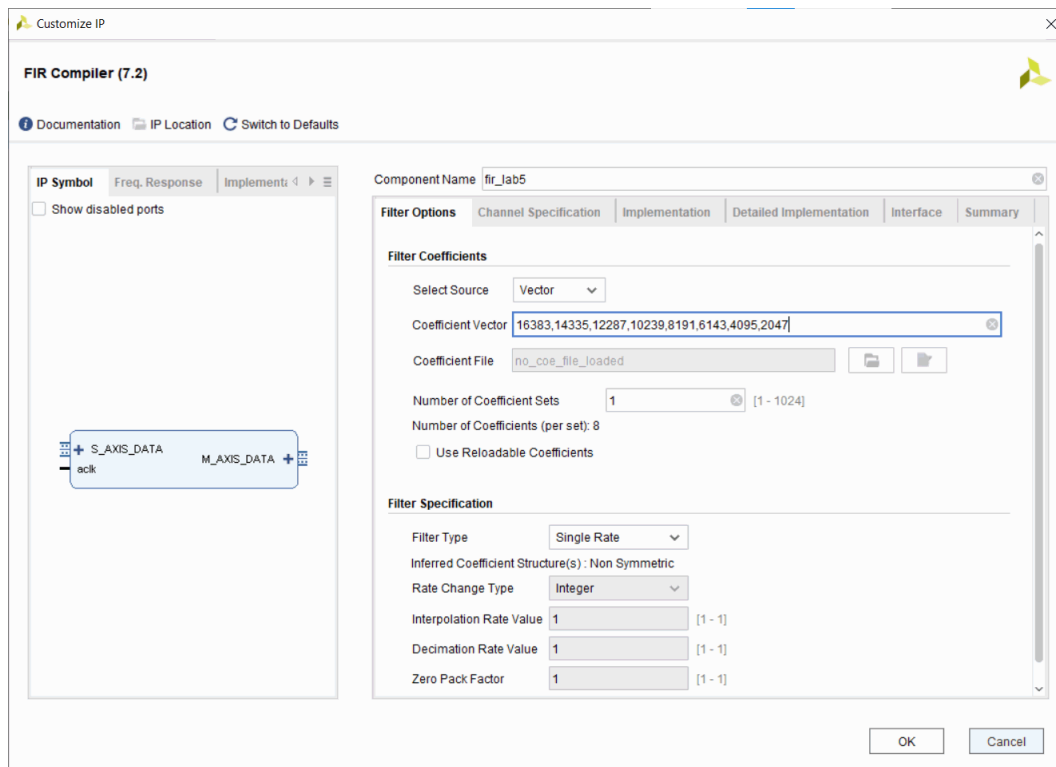
Po wykonaniu tych operacji środowisko Vivado przez kilka sekund analizuje pliki i tworzy nowy projekt.

Jeśli wszystkie pliki źródłowe są już dodane do projektu, to w oknie pokazującym jego strukturę (*Hierarchy*) nie powinny pojawiać się znaki zapytania przy modułach. Moduł nadrzędny (tutaj: `fpgatop`) jest umieszczony na górze i oznaczony ikoną z trzema kwadracikami – patrz rys. 4.5. Utworzony w ten sposób projekt jest spójny i gotowy do przeprowadzenia syntezy logicznej, implementacji i uruchomienia w układzie FPGA. Na potrzeby realizacji tego ćwiczenia, wymagane są jednak dodatkowe modyfikacje projektu.

5.5.2 Utworzenie bloku filtracji FIR

W oknie *Flow Navigator* (po lewej stronie) należy kliknąć na pozycji *IP Catalog* (czwarta pozycja od góry). Następnie w katalogu modułów IP wyszukujemy folder *Digital Signal Processing* a w nim pozycję *Filters* – patrz rys. 5.1. Ostatecznie wybieramy (dwukrotnie klikając) kreator *FIR Compiler*. Otworzy się okno *Customize IP* kreatora modułu FIR wybranego z biblioteki *IP Catalog*, pokazane na rys. 5.9.

W polu nazwy komponentu (*Component Name*) wpisujemy: `fir_lab5`. W pierwszej zakładce ustawień filtru (*Filter Options*) wpisujemy:



Rysunek 5.9: Pierwsza zakładka ustawień filtru FIR

- źródło współczynników filtru: *Select Source: Vector*,
- wektor współczynników *Coefficient Vector*⁶:
16383, 14335, 12287, 10239, 8191, 6143, 4095, 2047
- liczba zbiorów współczynników: *Number of Coefficient Sets: 1*,
- typ filtru: *Filter Type: Single Rate*.

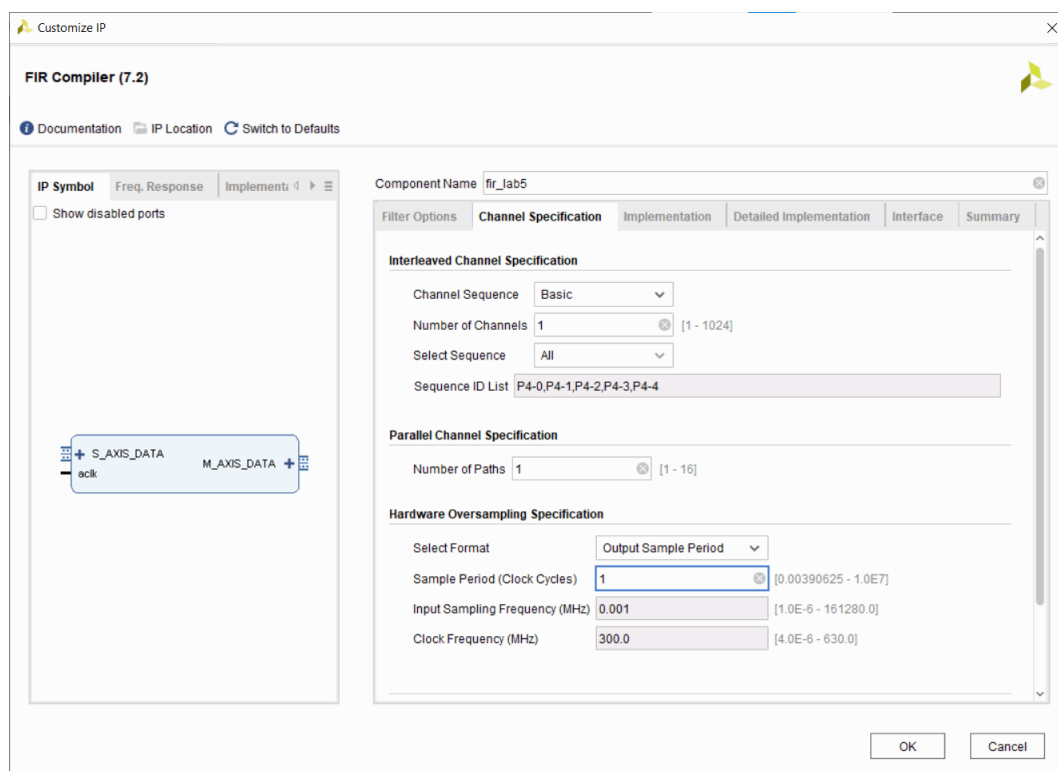
Po wprowadzeniu tych ustawień można obejrzeć charakterystykę częstotliwościową naszego filtru w zakładce *Frequency Response* po lewej stronie okna kreatora. Przechodzimy następnie do drugiej zakładki ustawień filtru *Channel Specification*.

W drugiej zakładce ustawień filtru, w górnej jej części ustawienia powinny być jak na rys. 5.10). W dolnej części zakładki (*Hardware Oversampling Specification*) wybieramy:

- *Select Format: Output Sample Period*,
- *Sample Period (Clock Cycles): 1*.

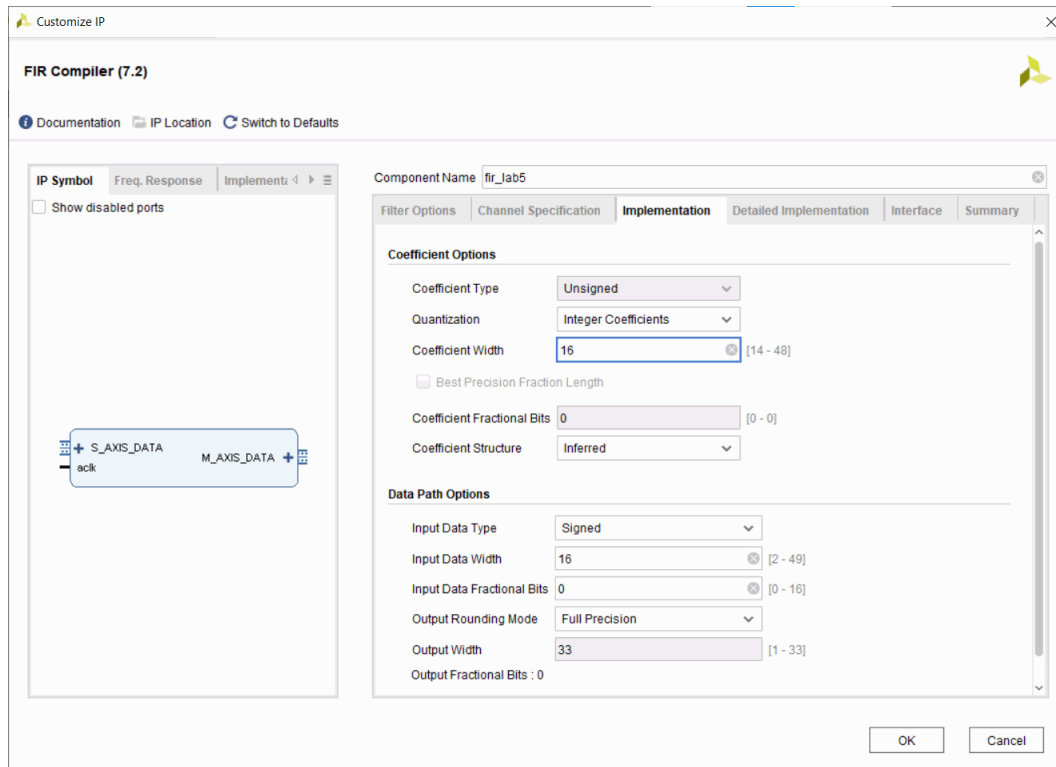
W ten sposób wymuszamy taką realizację filtru, która w każdym cyklu zegara będzie w stanie wystawiać na wyjście kolejny wynik obliczeń.

⁶Najlepiej skopiować (ctrl V) cały zestaw współczynników, zamiast ręcznego przepisywania.



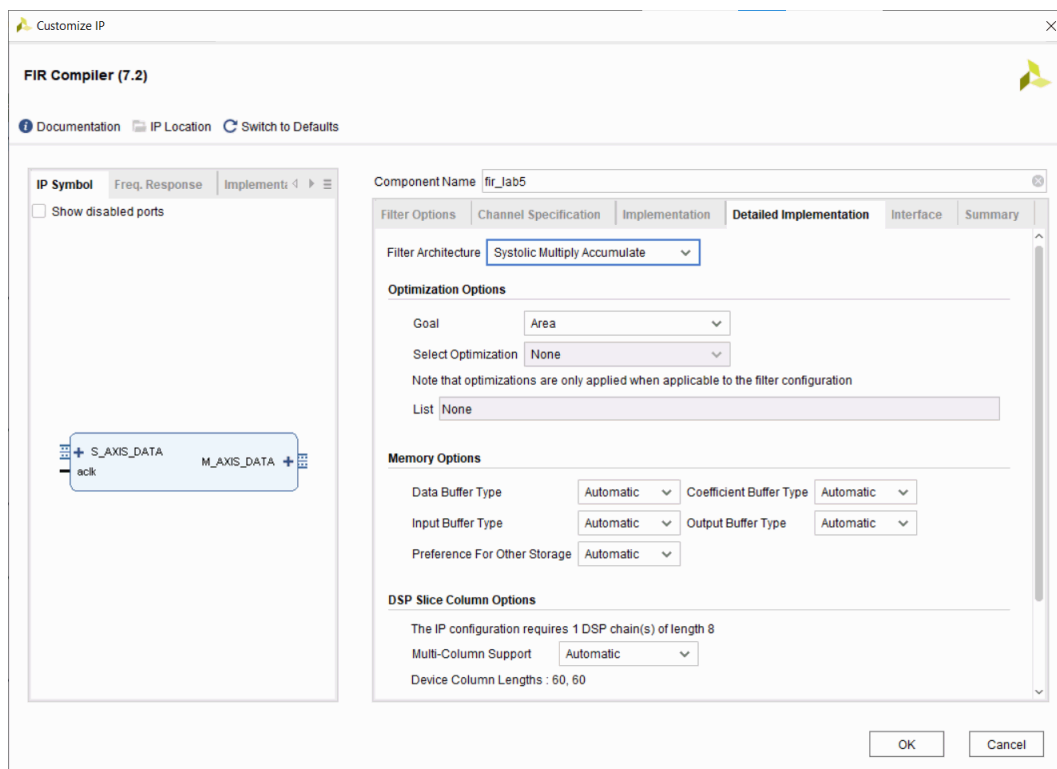
Rysunek 5.10: Druga zakładka ustawień filtru FIR

W trzeciej zakładce ustawień (*Implementation*) nic nie zmieniamy – ustawienia powinny być jak na rys. 5.11).



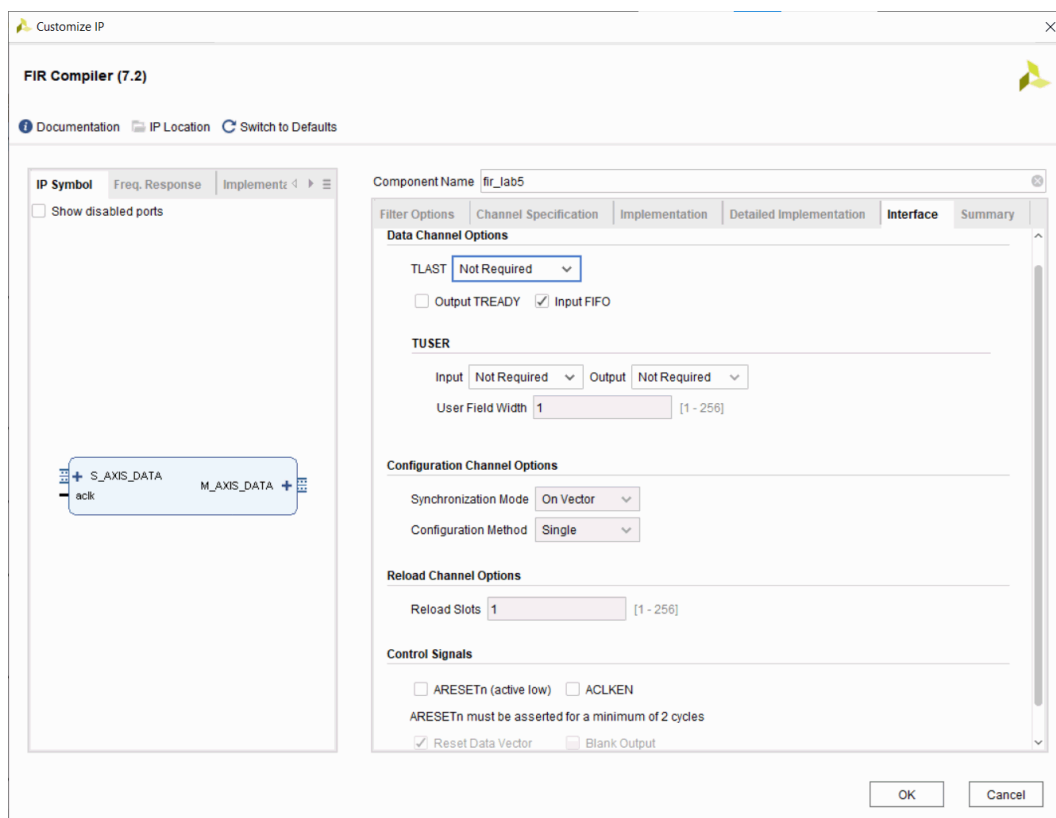
Rysunek 5.11: Trzecia zakładka ustawień filtru FIR

W czwartej zakładce ustawień (*Detailed Implementation*) ustawiamy architekturę filtru na *Systolic Multiply Accumulate* – jak na rys. 5.12).



Rysunek 5.12: Czwarta zakładka ustawień filtru FIR

W piątej zakładce (*Interface*) pozostawiamy ustawienia jak na rys. 5.13.



Rysunek 5.13: Piąta zakładka ustawień filtru FIR

Przechodzimy do ostatniej, szóstej zakładki z podsumowaniem ustawień filtra (*Summary*). Dodatkowo w lewej części okna kreatora można wybrać zakładkę *Implementation Details*, w której pokazana będzie zajętość kluczowych zasobów FPGA oraz parametry czasowe naszego modułu (w polu *Information*), jak na rys. 5.14. **Parametry te notujemy w formacie sprawozdania** do późniejszego porównania z wynikami uruchomienia na sprzęcie.

FIR Compiler (7.2)

Documentation IP Location Switch to Defaults

Component Name: `fir_lab5`

Implementation Details

Resource Estimates

DSP slice count	8
BRAM count	0

Information

Start-up Latency:	14
Calculated Coefficients:	8
Coefficient front padding:	0
Processing cycles per output:	1

AXI4 Stream Port Structure

S_AXIS_DATA - TDATA

Transaction	Field	Type
0	REAL(15:0)	fix16_0

M_AXIS_DATA - TDATA

Transaction	Field	Type
0	REAL(32:0)	fix33_0

Interleaved Channel Sequences

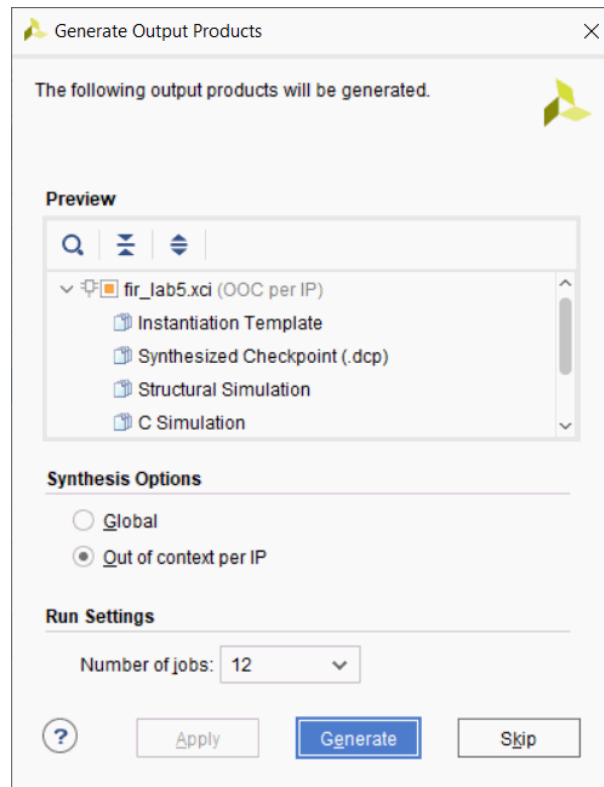
Filter Options	Channel Specification	Implementation	Detailed Implementation	Interface	Summary
Component Name	fir_lab5				
Filter Type	Single Rate				
Number of Interleaved Channels	1				
Number of Parallel Channels	1				
Clock Frequency	N/A				
Input Sampling Frequency	N/A				
Sample Period	1				
Number of Parallel Input Samples	1				
Number of Parallel Output Samples	1				
Input Data Width	16				
Input Data Fractional Bits	0				
Number of Coefficients	8				
Calculated Coefficients	8				
Number of Coefficient Sets	1				
Reloadable Coefficients	No				
Coefficient Structure	Non Symmetric				
Coefficient Width	16				
Coefficient Fractional Bits	0				
Quantization Mode	Integer_Coefficients				
Filter gain due to quantization mode	N/A				
Rounding Mode	Full Precision				
Output Width	33 (full precision = 33 bits)				
Output Fractional Bits	0				
Cycle Latency	14				
Filter Architecture	Systolic Multiply Accumulate				

OK Cancel

Rysunek 5.14: Szósta zakładka – podsumowanie ustawień filtra FIR

Po wprowadzeniu wszystkich ustawień modułu FIR zgodnie z powyższą instrukcją, klikamy na przycisku OK w prawej dolnej części okna kreatora.

W oknie *Create directory* zgadzamy się, klikając na przycisku OK.



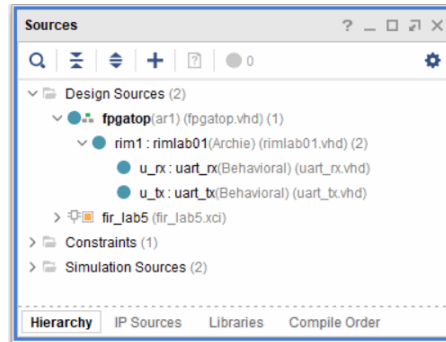
Rysunek 5.15: Końcowe okno generacji modułu FIR

W końcowym oknie *Generate Output Products* (patrz rys. 5.15) klikamy na przycisku *Generate*.

Okno kreatora ostatecznie zamyka się i wracamy do środowiska projektowego.

5.5.3 Instalowanie komponentu FIR w projekcie

Po wygenerowaniu komponentu FIR, w drzewie źródeł naszego projektu powinien pojawić się blok IP o nazwie jaką ustaliliśmy na samym początku (fir_lab5), tak jak pokazano na rys. 5.16. Początkowo ten blok nie jest powiązany z głównym plikiem projektu, co będziemy zmieniać w tym etapie pracy.



Rysunek 5.16: Źródła projektu po dodaniu bloku IP z modułem FIR

W podkatalogu `folder_projektu.srcs/sources_1/ip/fir/_lab5` znajdują się wygenerowane automatycznie źródła nowego komponentu IP, między innymi szablony `fir_lab5.vho`, który pokazuje prawidłowy sposób wykorzystania tego komponentu w naszym kodzie źródłowym VHDL. W głównym pliku naszego projektu (`fpgatop.vhd`) należy umieścić **deklarację komponentu** `fir_lab5` (skopiowaną z szablonu `fir_lab5.vho`):

```
COMPONENT fir_lab5
PORT (
    aclk : IN STD_LOGIC;
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    m_axis_data_tvalid : OUT STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(39 DOWNTO 0)
);
END COMPONENT;
```

W pliku `fpgatop.vhd` trzeba również zadeklarować dodatkowe sygnały do komunikacji z komponentem `fir_lab5`:

```
signal FIR_RFD : STD_LOGIC;
signal FIR_DOUT : STD_LOGIC_VECTOR (39 downto 0);
```

W opisie architektury (po słowie kluczowym **BEGIN**, ale nie w obrębie procesu !) umieszczamy natomiast *instancję* naszego nowego komponentu:

```
filtr_1 : fir_lab5
PORT MAP (
    aclk => clk,
    s_axis_data_tvalid => SRC_RD,
    s_axis_data_tready => FIR_RFD,
    s_axis_data_tdata => SRC_DATA,
```

```

    m_axis_data_tvalid => RESULT_WR,
    m_axis_data_tdata  => FIR_DOUT
);

```

W tym samym miejscu w opisie architektury umieszczamy także instrukcje współbieżne, czyli np. przypisania i połączenia sygnałów. Sygnał odczytu pamięci danych SRC_RD będzie teraz generowany na podstawie flag gotowości modułu FIR oraz naszego algorytmu:

```
SRC_RD <= FIR_RFD and PROC_WORKING;
```

Jako wynik końcowy obliczeń należy wybrać tylko 16 bitów z całego słowa danych na wyjściu modułu FIR:

```
RESULT_DATA <= FIR_DOUT (32 downto 17);
```

W procesie `fill_mem`, opisującym automat stanu naszego projektu, należy usunąć (lub zakomentować) wszystkie przypisania do sygnałów: `RESULT_WR`, `SRC_RD`, `RESULT_DATA`, które będą teraz sterowane w innych miejscach kodu. Ponadto należy ograniczyć działanie procesu tylko do dwóch stanów: `IDLE`, `RD_DAT`. Po tych zmianach opis procesu wygląda następująco:

```

-- Automat do sterowania pracą komponentu IP
fill_mem: process (RST, clk)
begin
    if RST='1' then -- zerowanie automatu
        stan_alg <= IDLE;
        PROC_WORKING<='0';
        cnt_data<=0;
    elsif rising_edge(clk) then
        if stan_alg=IDLE then -- oczekiwanie na sygnał START
            if START_PROC='1' then -- Uruchomienie pętli algorytmu
                PROC_WORKING<='1'; -- sygnalizacja pracy alg.
                cnt_data<=0;
                stan_alg<=RD_DAT;
            else -- oczekiwanie na sygnał START w bezczynności
                PROC_WORKING<='0';
            end if;
        elsif stan_alg=RD_DAT then -- Odczytanie słowa z pamięci
            if RESULT_WR='1' then -- jeśli wpisano słowo do wyników
                if cnt_data=1023 then -- czy koniec pracy algorytmu?

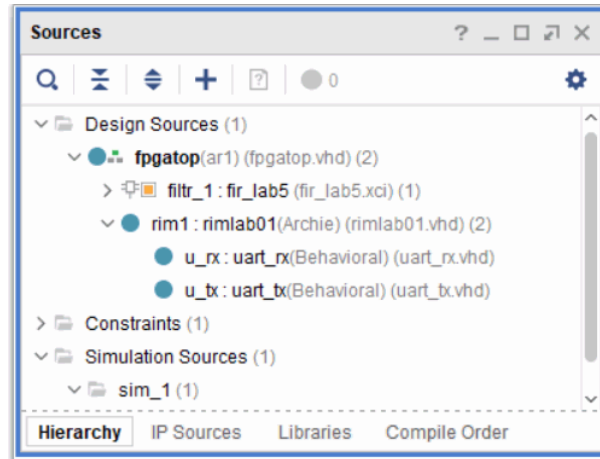
                    stan_alg<=IDLE;
                    PROC_WORKING<='0';
                else cnt_data<=cnt_data+1; -- następne słowo danych
                    stan_alg<=RD_DAT;
                end if; -- if cnt_data
            end if; -- RESULT_WR
        else stan_alg <= IDLE;
        end if; -- stan
    end if; -- RST/CLK

```

```
end process;
```

5.5.4 Kompilowanie projektu

Jeśli wszystkie elementy projektu zostały opisane i połączone prawidłowo, to drzewo projektu powinno wyglądać tak jak pokazano na rys. 5.17.



Rysunek 5.17: Źródła projektu z dołączonym komponentem fir_lab5

Jeśli edytor kodu VHDL nie sygnalizuje na czerwono błędów w pliku `fpgatop.vhd`, należy przeprowadzić implementację projektu, zgodnie z opisem w p. 4.3.3. Jeśli proces zakończy się bez błędów, można przesłać konfigurację do układu tak jak opisano w p. 4.3.4. Jeśli natomiast wystąpiły błędy, należy je usunąć i powtórzyć proces implementacji.

5.5.5 Badanie zrównoleglonej architektury MAC

W poprzednim punkcie utworzyliśmy komponent filtru FIR w strukturze bezpośredniej (*Systolic Multiply Accumulate*), wykorzystującej bloki mnożące do realizacji operacji mnożenia z akumulacją (MAC). Komponent ten użyliśmy w naszym projekcie.

Po przesłaniu konfiguracji do układu FPGA uruchamiamy środowisko Matlab, a następnie odpowiednio zmieniamy katalog bieżący oraz przygotowujemy ciąg danych wejściowych dla naszego algorytmu filtracji:

```
cd c:\rim\fpga
x=zeros(1,1024);
x(32:16:1008)=32767;
x(24:36)=32767;
```

Dane testowe w wektorze `x` zawierają na początku sygnał schodkowy, a następnie okresowo powtarzające się impulsy. Po przygotowaniu danych uruchamiamy pracę naszego algorytmu filtracji i rysujemy dane wejściowe i wynikowe na wspólnym wykresie:

```
y=fpga(x);
t=0:1023;
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 100 0 33000]); grid;
```

Na wykresie będzie widać: na czarno – impulsy pobudzające, oraz na czerwono – odpowiedź filtru. Wykres warto zapisać na dysku w celu porównania z następnymi realizacjami:

```
print -dpng fir-mac-1.png
```

Jeśli algorytm prawidłowo zakończył pracę i wykres pokazał się w środowisku Matlab, należy odpowiedzieć na poniższe pytania. W celu dokładniejszego zbadania pracy algorytmu można wykorzystać własną konstrukcję sygnału pobudzającego x (np. inne rozmieszczenie impulsów, dodatkowe impulsy schodkowe, itp.)

1. Jaka jest amplituda odpowiedzi na impuls i sygnał schodkowy ? Czy dochodzi do przepełnienia?
2. Jakie jest przesunięcie na wykresie odpowiedzi względem pobudzenia wejściowe ?
3. Ile cykli zegarowych trwała praca algorytmu ? Ile z tych cykli zajęła inicjacja filtru ?
4. Ile cykli trwało obliczanie każdego słowa na wyjściu ?
5. Porównać powyższe parametry czasowe z estymatami dostępnymi podczas tworzenia modułu (rys. 5.14)
6. Ile zasobów logicznych (LUT, FF) oraz bloków DSP w układzie FPGA wykorzystuje ta implementacja ?
7. Ile czasu trwała implementacja projektu (*Design Runs: synth_1/impl_1 - Elapsed*) ?

5.5.6 Badanie sekwencyjnej architektury MAC

W drzewie projektu odnajdujemy wygenerowany komponent IP `filtr_1 - fir_lab5` (patrz rys. 5.17) i dwa razy klikamy na tej pozycji. W ten sposób ponownie otworzy się program *Core Generator*.

Przechodzimy do drugiej zakładki ustawień filtru *Channel Specification*. Modyfikujemy okres próbkowania (*Sample Period*) z 1 na 8 cykli zegara. Zatwierdzamy zmianę przyciskiem OK. W nowym oknie *Generate Output Products* zamawiamy nową wersję komponentu przyciskiem *Generate*.

Po zakończeniu procesu generacji bloku FIR zamykamy okno *Generate Output Products* przyciskiem OK. Nowo wygenerowany komponent posiada identyczny zestaw portów jak ten w poprzedniej wersji, nie są więc wymagane żadne zmiany w kodzie źródłowym VHDL reszty naszego projektu. Uruchamiamy kompilację projektu, klikając na opcji *Generate Bitstream* (na końcu okna *Flow Navigator*), lub tak jak opisano w p. 5.5.4. Na koniec programujemy układ FPGA z nową konfiguracją, wykorzystując opcję *Open Hardware Manager*.

W środowisku Matlab przygotowujemy testowy ciąg danych dla algorytmu filtracji:

```
x=zeros(1,1024);
x(32:16:1008)=32767;
x(24:36)=32767;
```

Uruchamiamy pracę naszego algorytmu filtracji i rysujemy wykres z danymi:

```
y=fpga(x);
t=0:1023;
plot(t,x,'--k',t,y,'-r'); legend('wejście','wyniki');
axis([0 100 0 33000]); grid;
```

Wykres znów warto zapisać na dysku w celu porównania z kolejną realizacją:

```
print -dpng fir-mac-8.png
```

Jeśli algorytm prawidłowo zakończył pracę i wykres pokazał się, należy odpowiedzieć na poniższe pytania:

1. Czy zmieniła się amplituda odpowiedzi na impuls i sygnał schodkowy?
2. Jakie jest przesunięcie na wykresie odpowiedzi względem pobudzenia na wejściu ?
3. Ile cykli trwało obliczanie jednej próbki wyjściowej ?
4. Ile cykli zegarowych trwała praca algorytmu ? Ile z tych cykli zajęła inicjacja filtru ?
5. Porównać powyższe parametry czasowe z estymatami dostępnymi podczas tworzenia modułu (rys. 5.14)
6. Jak zmieniła się zajętość zasobów logicznych (LUT/FF) oraz bloków DSP w układzie FPGA ?
7. Ile czasu trwała implementacja projektu ? Porównać z poprzednią implementacją.

5.6 Badanie implementacji transformaty Fouriera

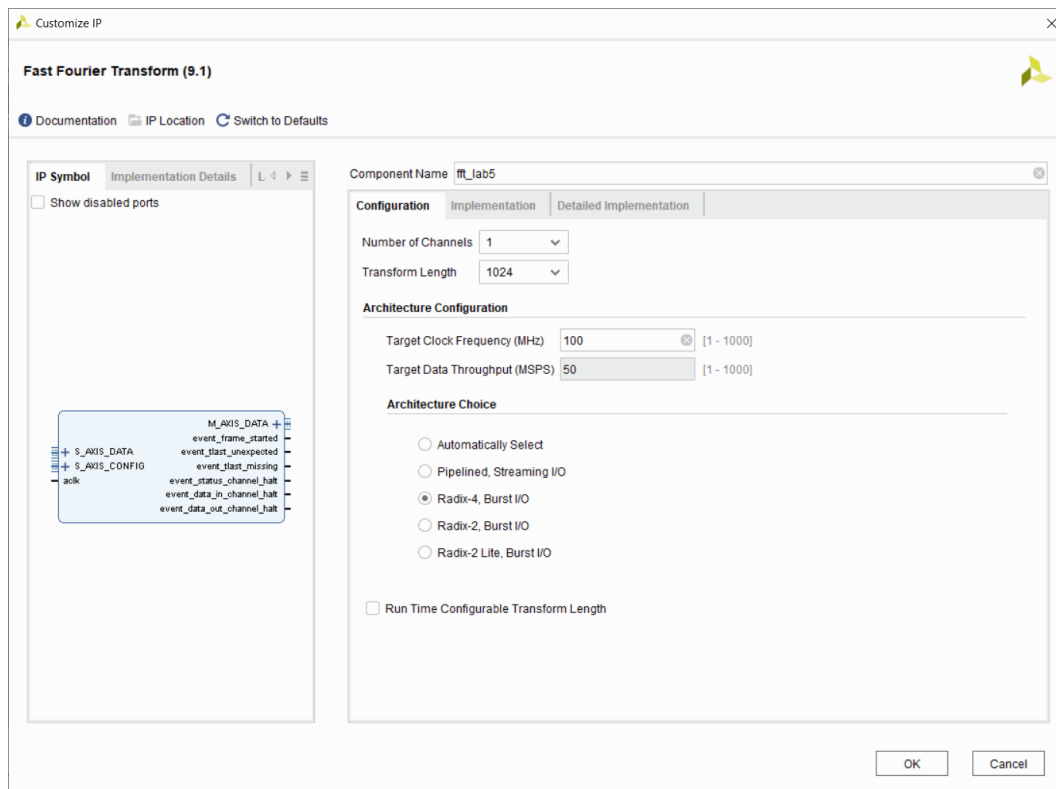
5.6.1 Utworzenie bloku FFT

Przejdziemy do generacji bloku IP realizującego transformatę FFT. W oknie *Flow Navigator* (po lewej stronie) klikamy na pozycji *IP Catalog*. Następnie w katalogu modułów IP wyszukujemy folder *Digital Signal Processing* a w nim pozycję *Transforms* i dalej: *FFTs*. Ostatecznie wybieramy kreator *Fast Fourier Transform*, dwukrotnie klikając. Otworzy się okno *Customize IP* kreatora modułu FFT, pokazane na rys. 5.18.

W polu nazwy komponentu (*Component Name*) wpisujemy: `fft_lab5`. W pierwszej zakładce (*Configuration*) główne ustawienia:

- liczba kanałów (*Number of Channels*): 1,
- rozmiar transformaty *Transform Length*: 1024,
- częstotliwość taktowania: *Target Clock Frequency*: 100 MHz,
- typ architektury: *Architecture Choice*: *Radix-4, Burst I/O*.

Przechodzimy do drugiej zakładki (*Implementation*), w której wybieramy naturalną kolejność próbek widma na wyjściu transformaty: *Output Ordering* : *Natural Order* i pozostawiamy resztę ustawień tak jak pokazano na rys. 5.19.

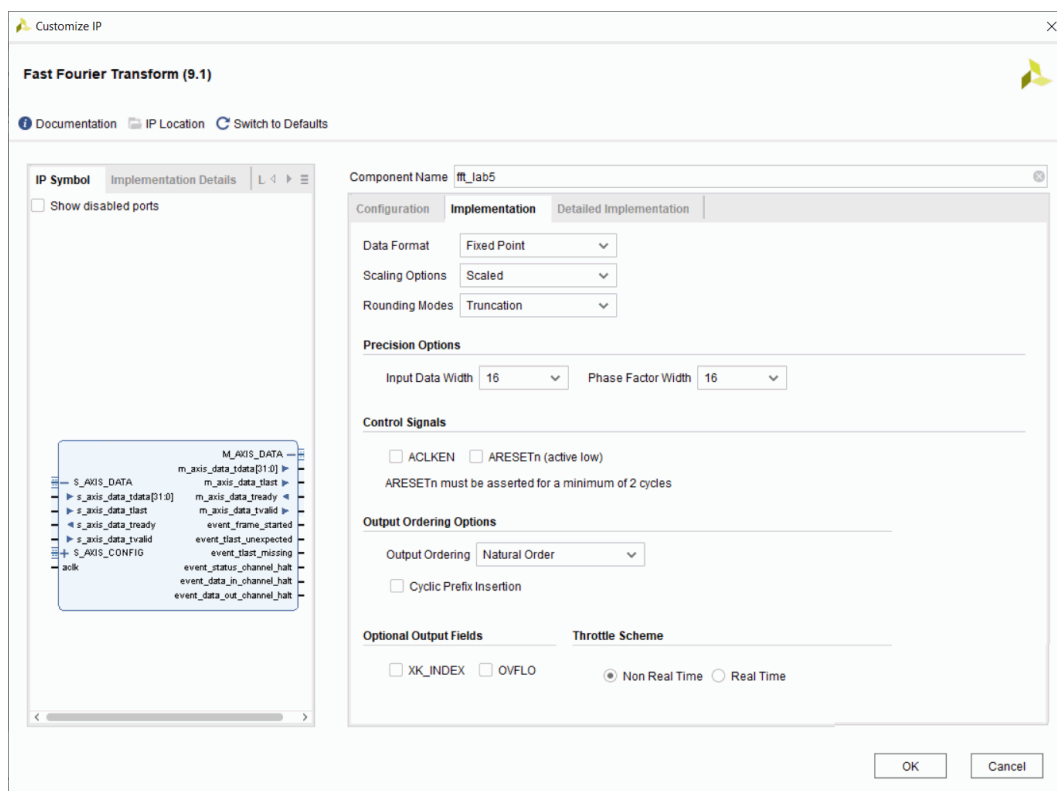


Rysunek 5.18: Pierwsza zakładka konfiguracji bloku FFT

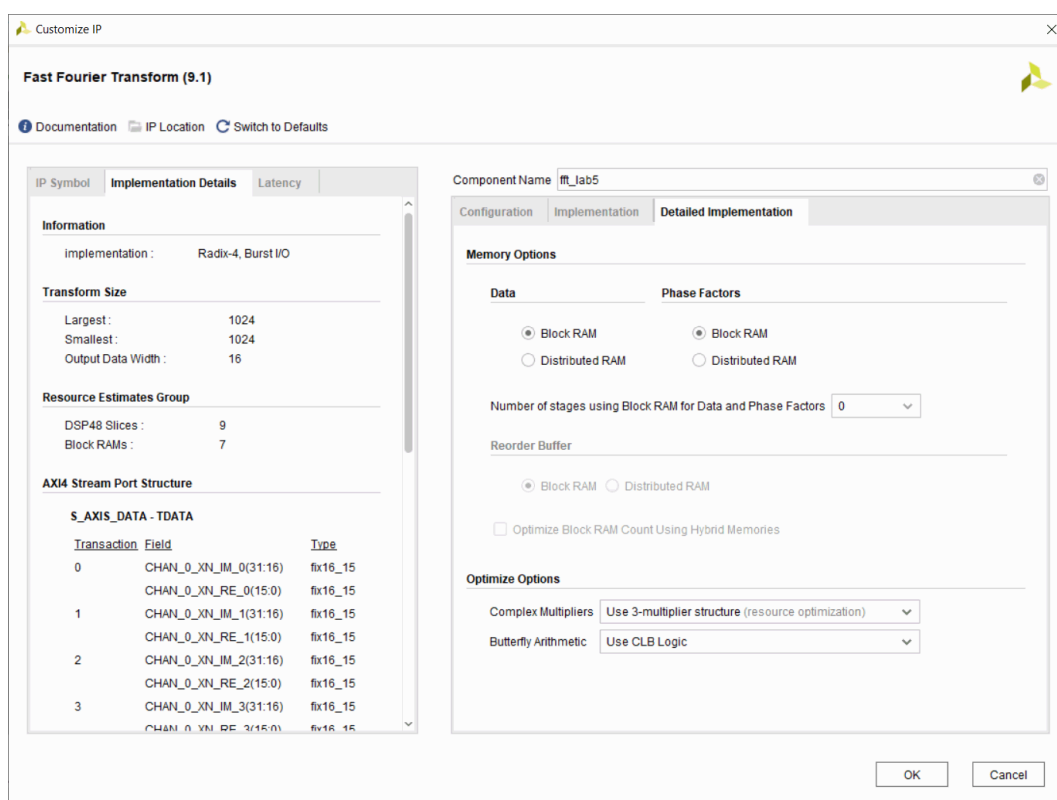
Przechodzimy do trzeciej zakładki (*Detailed Implementation*), w której ustawienia powinny wyglądać jak na rys. 5.20 (nic nie zmieniamy).

Na koniec klikamy na przycisku OK w prawej dolnej części okna kreatora. W oknie *Create directory* zgadzamy się, klikając na przycisku OK. W końcowym oknie *Generate Output Products* (patrz rys. 5.15) klikamy na przycisku *Generate*. Rozpoczyna się synteza logiczna naszego modułu FFT, która może trwać do dwóch minut. Okno kreatora ostatecznie zamyka się i wracamy do środowiska projektowego.

Po zakończeniu kompilacji, w podkatalogu `folder_projektu.srcs/sources_1/ip/fir/_lab5` znajdują się wygenerowane automatycznie źródła nowego komponentu IP. Jest wśród nich szablon `fft_lab5.vho`, który pokazuje prawidłowy sposób wykorzystania komponentu w naszym kodzie źródłowym VHDL.



Rysunek 5.19: Druga zakładka konfiguracji bloku FFT



Rysunek 5.20: Trzecia zakładka konfiguracji bloku FFT

5.6.2 Instalowanie bloku FFT w projekcie

W głównym pliku projektu (`fpgatop.vhd`), w odp. miejscu opisu architektury należy umieścić **deklarację komponentu** `fft_lab5` skopiowaną z szablonu `fft_lab5.vho`:

```
COMPONENT fft_lab5
PORT ( aclk : IN STD_LOGIC;
        s_axis_config_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        s_axis_config_tvalid : IN STD_LOGIC;
        s_axis_config_tready : OUT STD_LOGIC;
        s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        s_axis_data_tvalid : IN STD_LOGIC;
        s_axis_data_tready : OUT STD_LOGIC;
        s_axis_data_tlast : IN STD_LOGIC;
        m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        m_axis_data_tvalid : OUT STD_LOGIC;
        m_axis_data_tready : IN STD_LOGIC;
        m_axis_data_tlast : OUT STD_LOGIC;
        event_frame_started : OUT STD_LOGIC;
        event_tlast_unexpected : OUT STD_LOGIC;
        event_tlast_missing : OUT STD_LOGIC;
        event_status_channel_halt : OUT STD_LOGIC;
        event_data_in_channel_halt : OUT STD_LOGIC;
        event_data_out_channel_halt : OUT STD_LOGIC
    );
END COMPONENT;
```

Następnie definiujemy kilka nowych sygnałów potrzebnych do połączenia z modułem FFT:

```
signal FFT_RFD : STD_LOGIC;
signal FFT_DIN, FFT_DOUT : STD_LOGIC_VECTOR (31 downto 0);
signal FFT_RE, FFT_IM : STD_LOGIC_VECTOR (15 downto 0);
```

Dalej w opisie architektury, już po słowie `BEGIN` wstawiamy nasz nowy moduł `fft_top` i dołączamy jego porty do sygnałów dostępnych w projekcie (pomijając nieużywane wyjścia):

```
blok_fft : fft_lab5
PORT MAP (
    aclk => CLK,
    s_axis_config_tdata => (others => '0'),
    s_axis_config_tvalid => '0',
    s_axis_config_tready => open,

    s_axis_data_tdata => FFT_DIN,
    s_axis_data_tvalid => SRC_RD,
    s_axis_data_tready => FFT_RFD,
    s_axis_data_tlast => '0',

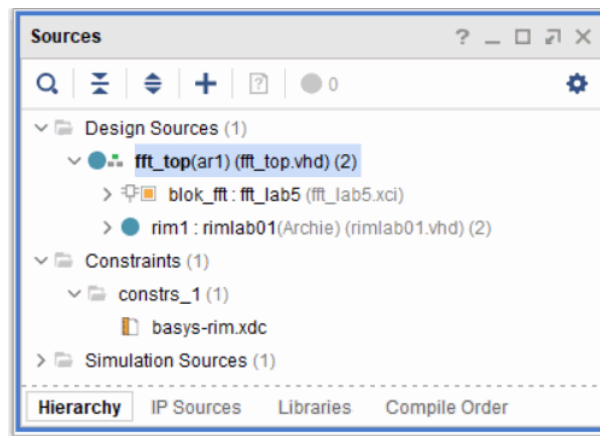
    m_axis_data_tdata => FFT_DOUT,
    m_axis_data_tvalid => RESULT_WR,
    m_axis_data_tready => PROC_WORKING,
    m_axis_data_tlast => open
);
```

Następnie definiujemy sygnały sterujące i wybieramy dane wejściowe oraz wyjściowe dla modułu FFT:

```
SRC_RD  <= FFT_RFD and PROC_WORKING;
FFT_DIN <= X"0000"&SRC_DATA; -- dane wejściowe: rzeczywiste
FFT_RE  <= FFT_DOUT(15 downto 0); -- część rzeczywista widma
FFT_IM  <= FFT_DOUT(31 downto 16); -- część urojona widma
RESULT_DATA <= FFT_RE; -- dane wysyłane jako wynik obliczeń
```

Należy także pamiętać o usunięciu poprzednio badanego komponentu filtru FIR (`filtr_1 : fir_lab5`), a przede wszystkim związanych z nim przypisań do sygnałów: `SRC_RD`, `RESULT_DATA`. W innym przypadku kompilacja projektu nie będzie możliwa ze względu na kolizje na tych sygnałach.

Jeżeli komponent FFT został prawidłowo dołączony do kodu VHDL naszego projektu, hierarchia źródeł (*Sources*) powinna wyglądać tak jak pokazano na rys. 5.21.



Rysunek 5.21: Źródła projektu z dołączonym komponentem `fft_lab5`

5.6.3 Kompilowanie projektu

Jeśli na prawym marginesie okna edycji naszego kodu VHDL nie pojawiają się czerwone sygnały o błędach, a na górze marginesu wyświetla się zielone pole statusu – możemy rozpocząć kompilację całego projektu. W tym celu możemy od razu kliknąć na pozycji *Generate Bitstream* na dole okna *Flow Navigator*.

Potwierdzamy nasze intencje przyciskiem *Yes* w wyskakującym oknie *No implementation Results Available* i następnie klikamy na *OK* w oknie *Launch Runs*. Teraz środowisko automatycznie przejdzie przez wszystkie etapy kompilacji projektu (syntezę logiczną, implementację, generację *bitstreamu*), co może potrwać do dwóch minut.

5.6.4 Badanie algorytmu FFT o podstawie 4

Po przesłaniu nowej konfiguracji do układu FPGA, wykorzystując opcję *Open Hardware Manager*, przechodzimy do środowiska Matlab. W razie potrzeby odpowiednio zmieniamy katalog bieżący:

```
cd c:\rim\fpga
```

a następnie uruchamiamy skrypt `fftttest.m` sprawdzający działanie algorytmu FFT w układzie FPGA:



```
% Skrypt do testowania algorytmu FFT
F1=0.26; % częstotliwość unormowana
F2=0.21; % częstotliwość unormowana
F3=0.16; % częstotliwość unormowana
F4=0.11; % częstotliwość unormowana

t=0:1023;
x=8000*sin(t*2*pi*F1);
%x=x+8000*cos(t*2*pi*F2);
%x=x+8000*cos(t*2*pi*F3);
%x=x+8000*cos(t*2*pi*F4);

y=fpga(x);
t=0:1023;
figure (1);
subplot (2,1,1);
plot (t,x);
axis ([0 1023 -20000 20000]);
title ('Przebieg sygnału wejściowego');
grid on;

subplot (2,1,2);
plot (t,y);
axis ([0 1023 -1500 3500]);
title ('Widmo obliczone w FPGA');
grid on;
```

Uzyskany wykres widma zapisujemy na dysku dla potrzeb porównania z kolejną wersją:

```
print -dpng fft-radix4.png
```

Testowanie algorytmu należy powtórzyć z większą liczbą składowych harmoniczných w sygnale wejściowym. Wykres widma obliczonego w układzie FPGA należy po drugim uruchomieniu skryptu `fftttest` porównać z widmem obliczonym w Matlabie za pomocą funkcji `fft()`:

```
w=fft(x,1024); figure (2); plot(real(w)); grid;
title('Widmo referencyjne z Matlab')

```

Jeśli algorytm zadziała prawidłowo, należy odpowiedzieć na poniższe pytania:

1. Jak zależy amplituda prążków widma od amplitudy sygnału na wejściu?
Jakim współczynnikiem skali są one związane?
2. Jakie są różnice w porównaniu z widmem obliczonym w Matlabie?
3. Ile cykli trwało obliczanie każdej próbki widma ?
4. Ile cykli zegarowych trwała praca algorytmu ?
Ile z tych cykli zajęła inicjacja bloku FFT ?

5. Jaka jest zajętość zasobów logicznych (LUT, FF) i bloków DSP ?
6. Ile czasu trwała implementacja projektu (*Design Runs: synth_1/impl_1 - Elapsed*) ?

5.6.5 Badanie algorytmu FFT o podstawie 2

Zmienimy architekturę bloku FFT na prostszą, wykorzystującą dekompozycję na „motylki” o podstawie 2. W tym celu, w drzewie projektu odnajdujemy nasz komponent IP `fft_lab5` (patrz rys. 5.21) i dwa razy klikamy na tej pozycji. W ten sposób ponownie otworzy się kreator dla bloku FFT.

W pierwszej zakładce *Configuration* zmieniamy wybór architektury (*Architecture Choice*) na: *Radix-2, Burst I/O*. Zatwierdzamy zmianę przyciskiem OK. W nowym oknie *Generate Output Products* zlecamy aktualizację komponentu przyciskiem *Generate*.

Synteza nowego modułu FFT trwa około 1,5 minuty. Po jej zakończeniu w oknie *Design Runs* widać będzie status: *Out-of-Context Module Runs / fft_lab5_synth_1 : synth_design Complete!*. Zamykamy okno *Generate Output Products* przyciskiem OK. W naszym kodzie VHDL trzeba dla formalności zaktualizować jedną linię w deklaracji komponentu `fft_lab5`:

```
s_axis_config_tdata:IN STD_LOGIC_VECTOR(23 DOWNT0 0); -- zamiast 15
```

Po wygenerowaniu nowej wersji bloku FFT, uruchamiamy kompilację projektu, klikając na opcji *Generate Bitstream* (na końcu okna *Flow Navigator*), lub tak jak opisano w p. 4.3.3. Na koniec programujemy układ FPGA z nową konfiguracją, wykorzystując opcję *Open Hardware Manager* (patrz p. 4.3.4).

W środowisku Matlab uruchamiamy skrypt `fftttest.m` testujący działanie algorytmu FFT w układzie FPGA, identyczny z tym z poprzedniego punktu. Wykres widma obliczonego w układzie FPGA należy po drugim uruchomieniu skryptu `fftttest` porównać z widmem obliczonym w Matlabie za pomocą funkcji `fft()`:

```
w=fft(x,1024); figure (2); plot(real(w)); grid;
title('Widmo referencyjne z Matlab')

```

a także z wykresem widma z poprzedniej implementacji, zapisanym do pliku `fft-radix4.png`.

Jeśli algorytm FFT zadziałał prawidłowo, należy odpowiedzieć na poniższe pytania:

1. Jak zależy amplituda prążków widma od amplitudy sygnału ? Jaki współczynnik skali należy przyjąć w tym przypadku ?
2. Jakie są różnice w porównaniu z widmem obliczonym w Matlabie ?
3. Jakie są różnice względem widma z poprzedniej realizacji (Radix-4) ?
4. Ile cykli trwało obliczanie każdej próbki widma ? Ile razy wzrósł ten czas, względem poprzedniej implementacji ?
5. Ile cykli zegarowych trwała praca algorytmu ? Ile z tych cykli zajęła inicjacja ?
6. Z czego wynika dłuższa praca algorytmu ?

7. Jaka jest teraz zajętość zasobów logicznych (LUT, FF) i bloków DSP ? Ilukrotny spadek obserwujemy, względem poprzedniej implementacji ?
8. Ile czasu trwała implementacja projektu (*Design Runs: synth_1/impl_1 - Elapsed*) ? Porównać z poprzednią implementacją.

5.6.6 Implementacja obliczeń modułu liczby zespolonej

W tym etapie, w głównym pliku naszego projektu (`fpgatop.vhd`), w opisie architektury (przed słowem `BEGIN`) należy umieścić **deklaracje komponentów** `mag16.vhd`, `delaysel.vhd` utworzonych samodzielnie w ramach przygotowań do ćwiczenia (zadania: 5.4.1, 5.4.2).

Dane wyjściowe z bloku FFT (tj. wektory: `FFT_RE`, `FFT_IM`) powinny trafiać do portów wejściowych `in_re`, `in_im` modułu `mag16` (przygotowanego w ramach pracy domowej – patrz p. 5.4.1), który ma za zadanie szybko obliczać przybliżenie modułu liczby zespolonej. Wynik tych obliczeń powinien być wpisywany jako słowo 16-bitowe do portu `DIN_RESULT` komponentu komunikacyjnego `rimlab01`, za pośrednictwem sygnału `RESULT_DATA`. Poprzednie przypisanie do tego sygnału, znajdujące się pod instancją komponentu `blok_fft`, należy zakomentować w celu uniknięcia kolizji.

Moduł `mag16` wymaga kilku cykli zegara do wykonania obliczeń. Aby wyrównać wprowadzane przez niego opóźnienie, sygnał gotowości danych do zapisu (`m_axis_data_tvalid`) powinien być opóźniany o taką samą liczbę cykli zegara w module `delaysel` (opisanym w p. 5.4.2), przed doprowadzeniem go do sygnału `RESULT_WR`. W tym w projekcie należy użyć modułu `delaysel` przygotowany w ramach pracy domowej i odpowiednio go podłączyć pomiędzy portem `m_axis_data_tvalid` modułu FFT, a sygnałem zapisu danych wynikowych: `RESULT_WR`. W tym celu konieczne będzie zdefiniowanie w projekcie dodatkowego sygnału pośredniczącego, np.:

```
signal fft_dv : STD_LOGIC;
```

i jego dołączenie do odpowiedniego wyjścia modułu FFT, zamiast sygnału `RESULT_WR`:

```
m_axis_data_tvalid => fft_dv,
```

oraz do portu wejściowego `din` modułu `delaysel`:

```
din => fft_dv,
```

Wartość opóźnienia, wynikającą z konkretnej implementacji modułu `mag16`, należy podać jako parametr `Generic sel` przy instancji komponentu `delaysel`.

Nasze nowe komponenty (`mag16.vhd`, `delaysel.vhd`) należy dodać do źródeł projektu w środowisku Vivado, klikając na ikonke „+” w oknie *Sources* i wskazując oba pliki źródłowe w odpowiednim katalogu.

Uruchamiamy kompilację projektu, klikając na opcji *Generate Bitstream* (na końcu okna *Flow Navigator*), lub tak jak opisano w p. 4.3.3. Na koniec programujemy układ FPGA z nową konfiguracją, wykorzystując do tego opcję: *Open Hardware Manager* (patrz p. 4.3.4).

W środowisku Matlab uruchamiamy ponownie skrypt `fftttest.m` testujący działanie algorytmu FFT w układzie FPGA, identyczny z tymi z poprzednich punktów. Wykres widma obliczonego w układzie FPGA należy porównać z widmem obliczonym w Matlabie za pomocą funkcji `fft()`, a także z wykresem z pierwszej implementacji, zapisanym do pliku `fft-radix4.png`.

Jeśli algorytm w układzie FPGA zadziałał prawidłowo, należy odpowiedzieć na poniższe pytania:

1. Jakie są różnice w widmie w porównaniu z obliczeniami w Matlabie ?
2. Ile cykli trwało obliczanie każdej próbki widma ?
3. Ile cykli zegarowych trwała praca algorytmu ? Ile z tych cykli zajęła inicjacja ?
4. Jaki jest koszt realizacji (LUT, FF, DSP) obliczania modułu liczby zespolonej ?

Uzyskane wyniki należy przedstawić do oceny prowadzącemu ćwiczenie.