SIMD – wrzucamy 6-ty bieg na CPU

SIMD (ang. Single Instruction, Multiple Data) jest typem architektury komputerowej, w której w danej chwili jedna instrukcja wykonywana jest jednocześnie na wielu strumieniach danych. Architektura ta jest charakterystyczna dla procesorów wektorowych i spotykana jest powszechnie w GPU kart graficznych. Nie jest jednak do nich ograniczona – dobrodziejstwa SIMD dostępne są również w procesorach z rodziny x86, ARM i innych, kryjąc się pod takimi nazwami marketingowymi, jak SSE, AVX czy NEON.

HISTORIA

Pierwszym komputerem, który wykonywał równoległe obliczenia na wielką skalę, był superkomputer ILLIAC IV. Wykorzystano w nim architekturę SIMD typu macierzowego, gdzie jednostka kontrolna (CU) zarządzała pracą sześćdziesięciu czterech elementów wykonawczych (PE). Każdy PE wyposażony był w pamięć zdolną pomieścić dwa tysiące 64-bitowych słów, a CU miała dostęp do wszystkich 128 tysięcy słów. Przy taktowaniu równym 25 MHz komputer miał osiągać wydajność na poziomie 1 GFLOPa. Ostatecznie budowa ILLIACa IV znacznie przekroczyła zarówno budżet, jak i termin zakończenia, a jego rzeczywista wydajność wynosiła ~10 MIPS. Choć projekt został ukończony, a komputer wykorzystywany był przez prawie 10 lat – do jego wyłączenia 7 września 1981 roku – uważa się go za porażkę. Niemniej jednak jego realizacja miała istotny wpływ na postęp w dziedzinach takich jak przetwarzanie równoległe i projektowanie wspomagane komputerowo, a także przyspieszenie upowszechnienia pamięci opartej o półprzewodniki.

W latach 90. XX wieku SIMD stopniowo wprowadzone zostało w procesorach komputerów osobistych.

JAK TO UGRYŹĆ

Nic nie stoi na przeszkodzie, aby implementować swoje rozwiązanie od razu przy pomocy SIMDów. Jednak zazwyczaj warto rozwiązać nasz problem najpierw w tradycyjny sposób – nazywany implementacją skalarną. Następnie analizując działający kod i dokumentację instrukcji procesora, możemy zacząć przekładać go na wersję wektorową. Warto w tym procesie wspierać się kartką papieru i czymś do pisania, aby ułatwić sobie pracę. Próba symulacji szeregu złożonych operacji na rejestrach wektorowych wyłącznie w głowie nie należy do rzeczy prostych i łatwo tutaj o błąd.

Jeśli nie zamierzamy pisać kodu bezpośrednio w asemblerze, musimy skorzystać z funkcji w języku C opartych o instrukcje SIMD. Funkcje te (ang. *intrinsic functions*), i odpowiednie dla nich typy danych, staną się dostępne po załączeniu odpowiednich plików nagłówkowych. Nie ma potrzeby instalowania żadnych dodatków czy rozszerzeń – popularne kompilatory takie jak GCC, Clang czy MSVC zapewniają wsparcie dla instrukcji SIMD architektury x86 i ARM. W Listingu 1 przedstawiono fragment kodu odpowiedzialny za załączenie odpowiedniego, w zależności od kompilatora i architektury, pliku nagłówkowego.

Listing 1. Załączanie odpowiednich plików nagłówkowych.

```
#ifdef _MSC_VER
    #include <intrin.h>
#else
#if defined __x86_64__ || defined __i386__
    #include <x86intrin.h>
#elif defined __ARM_NEON__
    #include <arm_neon.h>
#endif
#endif
```

Dokumentację intrinsiców Intela znajdziemy na stronie intel.com/content/www/us/en/docs/intrinsics-guide/index.html, zaś dla ARM – pod adresem developer.arm.com/architectures/instruction-sets/intrinsics/.

ANATOMIA SIMD

Nazwy funkcji, zarówno intelowskich, jak i ARM, zakodowane są jako mnemoniki, dzięki którym powinniśmy być w stanie szybko rozszyfrować ich przeznaczenie lub łatwo odszukać tę, której potrzebujemy. Oczywiście każdy dostawca architektury definiuje zarówno mnemoniki, jak i zbiór funkcji według własnego uznania. W związku z tym nie można oczekiwać, że jeden kod będzie działał na każdym sprzęcie. Konwersja z Intela na ARM, lub odwrotnie, zazwyczaj nie sprowadzi się jedynie do znalezienia odpowiedników funkcji z innej architektury. Przeważnie trzeba będzie, przynajmniej częściowo, zmodyfikować implementację, żeby uzyskać ten sam lub zbliżony efekt.

Mnemoniki Intela mają następującą postać:

```
_mm_<intrin_op>_<suffix>
```

gdzie:

- `<intrin_op>` – nazwa operacji, na przykład `add` (dodawanie), `sub` (odejmowanie), `mul` (mnożenie).
- `<suffix>` – typ danych, na jakich operuje dana funkcja. Pierwsza litera, lub dwie, mówi o tym, czy mamy do czynienia z danymi *packed* (`p`), *extended packed* (`ep`) czy skalarnymi (`s`). Pozostałe litery i cyfry wskazują, czy dane są zmiennoprzecinkowe o pojedynczej (`s`) lub podwójnej (`d`) precyzji, całkowite (`i`) lub naturalne (`u`). Wartość liczbowa określa, ile bitów przeznaczono na dane – 8, 16, 32, 64, 128, 256 lub 512.

Ponadto między `_mm_` a `<intrin_op>` możemy spotkać wartość 256 lub 512 – w przypadku instrukcji z zestawu AVX. Jest to dodatkowa informacja na temat wielkości rejestru, na którym operuje dana funkcja.

PRAWNO-AUTORSKA OCHRONA PROGRAMÓW KOMPUTEROWYCH I ELEMENTÓW Z NIMI ZWIĄZANYCH | STREFA CTF

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

6/2022 (105)
LISTOPAD / GRUDZIEŃ

Cena 28,90 zł (w tym VAT 8%)

ZAAWANSOWANE METODY DEBUGOWANIA W SYSTEMIE WINDOWS I VISUAL STUDIO

cppfront: rewolucyjny zwrot w historii języka C++?

Mierzenie metryk kodu C# przy pomocy bibliotek Roslyn

Podman – czy rzeczywiście bezpieczniej niż w Dockerze?

Zaawansowane techniki .NET MAUI

Od developera do architekta

Kompilacja programów z CMake

NOWY NUMER JUŻ W EMPIKACH

ISSN 2084-9400
06
9 772084 940206

Przykładowa funkcja:

```
__m128i _mm_max_epi16 (__m128i a, __m128i b)
```

porównuje 16-bitowe liczby całkowite w `a` i `b`, zapisując maksimum w `dst`.

Listing 2. Pseudokod funkcji _mm_max_epi16

```
FOR j := 0 to 7
    i := j*16
    dst[i+15:i] := MAX(a[i+15:i], b[i+15:i])
ENDFOR
```

Mnemoniki ARM mają następującą postać:

```
<opname>[q]_<type>
```

gdzie:

- `<opname>` – operacja, na przykład `add` (dodawanie), `sub` (odejmowanie), `mul` (mnożenie).
- `[q]` – operacja działa na 128-bitowych danych.
- `<type>` – typ danych wejściowych – `s` (ze znakiem) lub `u` (be znaku), następnie wartość liczbowa określająca wielkość w bitach.

Nazwa każdej operacji zaczyna się od litery v, po której następuje słowo opisujące jej charakter – na przykład dodawanie (`add`), odejmowanie (`sub`) czy mnożenie (`mul`). W jej skład może wchodzić również znak określający, czy mamy do czynienia z operacją wydłużającą (`l` – przetwarza dane wektorowe wielkości podwójnego słowa i zwraca dane wektorowe wielkości poczwórnego słowa), poszerzającą (`w` – przetwarza dane wektorowe wielkości podwójnego i poczwórnego słowa, zwracając dane wektorowe wielkości poczwórnego słowa) lub zawężającą (`n` – przetwarza dane wektorowe wielkości poczwórnego słowa i zwraca dane wektorowe wielkości podwójnego słowa).

Przykładowa funkcja:

```
int16x8_t vaddq_s16(int16x8_t a, int16x8_t b)
```

dodaje do siebie odpowiadające sobie 8-bitowe elementy wektora `a` oraz `b`, umieszczając rezultat w rejestrze wektorowym `d`.

Listing 3. Pseudokod funkcji vaddq_s16

```
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

ZASTOSOWANIE

Obszarem, w którym SIMDy dają bardzo dobre rezultaty, jest, między innymi, przetwarzanie obrazów. W dalszej części artykułu przedstawiony zostanie prosty program wykonujący splot danych bitmapy z filtrem, w rezultacie czego uzyskane zostaną efekty wyostrzenia i wykrycia krawędzi.

Zanim przejdziemy do analizy implementacji wersji skalarnej oraz wektorowej, przypomnijmy sobie, na czym polega operacja splatania (lub konwolucji) obrazu z filtrem (zwanym również jądrem bądź maską).

Najpierw definiujemy maskę w postaci macierzy o określonych wymiarach. W tym przypadku macierz 3x3.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Rysunek 1. Filtr tożsamościowy o wymiarach 3x3

Następnie nakładamy ją na dany region obrazu, mnożymy wartości pikseli przez odpowiadające im wartości z macierzy, sumujemy i rezultat zapisujemy jako wartość piksela obrazu wyjściowego. Ze względu na to, że maska okala piksel wejściowy, obraz wyjściowy będzie odpowiednio pomniejszony – odrzucamy wiersze i kolumny, dla których maska wykraczałaby poza obszar obrazu źródłowego.

7	8	9	3	2	…
7	7	4	2	5	…
6	5	5	3	3	…
4	4	2	2	2	…
3	5	5	1	3	…
⋮	⋮	⋮	⋮	⋮	

Rysunek 2. Kolorem czerwonym zaznaczono fragment obrazu, na który nakładany jest filtr o rozmiarze 3x3

W wyniku operacji splatania z filtrem tożsamościowym przedstawionym na Rysunku 2 otrzymujemy następujące równanie:

$$Pix_{out} = 0*7 + 0*8 + 0*9 + 0*7 + 1*7 + 0*4 + 0*6 + 0*5 + 0*5 = 7$$

Wspomniane wcześniej filtr wyostrzający oraz filtr Sobela przedstawiono kolejno na Rysunkach 3 i 4.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Rysunek 3. Maska filtru wyostrzającego

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Rysunek 4. Maska horyzontalnego filtru Sobela służącego do wykrywania krawędzi

IMPLEMENTACJA SKALARNA

Przedstawiony program jest w gruncie rzeczy dość prosty i sprowadza się do trzech kroków: załadowania pliku .png z obrazem, przeprowadzenia operacji splatania, zapisania wyniku do nowego pliku .png. Wczytania oraz zapisu obrazów dokonujemy za pomocą biblioteki libpng. Kod programu umieszczony został w serwisie GitHub, dlatego nie będziemy zagłębiać się tutaj w część odpowiedzialną za

czytanie i zapis pliku graficznego, skupiając się jedynie na fragmentach kodu odpowiedzialnych za operację splotu.

W wersji skalarnej najpierw, przy pomocy pętli, kopiujemy dane pikseli do tablicy o stałym rozmiarze.

Listing 4. Przygotowanie danych do pracy – kod skalarny

```
uint8_t pixCh[3][12] = { 0 };
uint32_t* tmp = curr;
for(int r = 0; r < 3; r++)
{
    tmp = curr + (r * w);
    memcpy(&pixCh[r][0], tmp, 12);
}
```

Następnie mnożymy odpowiednie wartości z tablicy z odpowiadającymi im wartościami z tablicy reprezentującej filtr splotowy, a wyniki kumulujemy w odpowiednich zmiennych dla poszczególnych kanałów – czerwonego, zielonego i niebieskiego.

Listing 2. Splatanie pikseli z filtrem

```
int rgb[3] = { 0 };
int idx = 0;
for(int row = 0; row < 3; row++)
{
    for(int c = 0; c < 12; c++)
    {
        int modulo = c % 4;
        if (modulo == 3)
        {
            ++idx;
        }
        else
        {
            rgb[modulo] += pixCh[row][c] * filter[idx];
        }
    }
}
```

Na końcu obcinamy wartości do zakresu 0-255 i zapisujemy je do bufora wyjściowego.

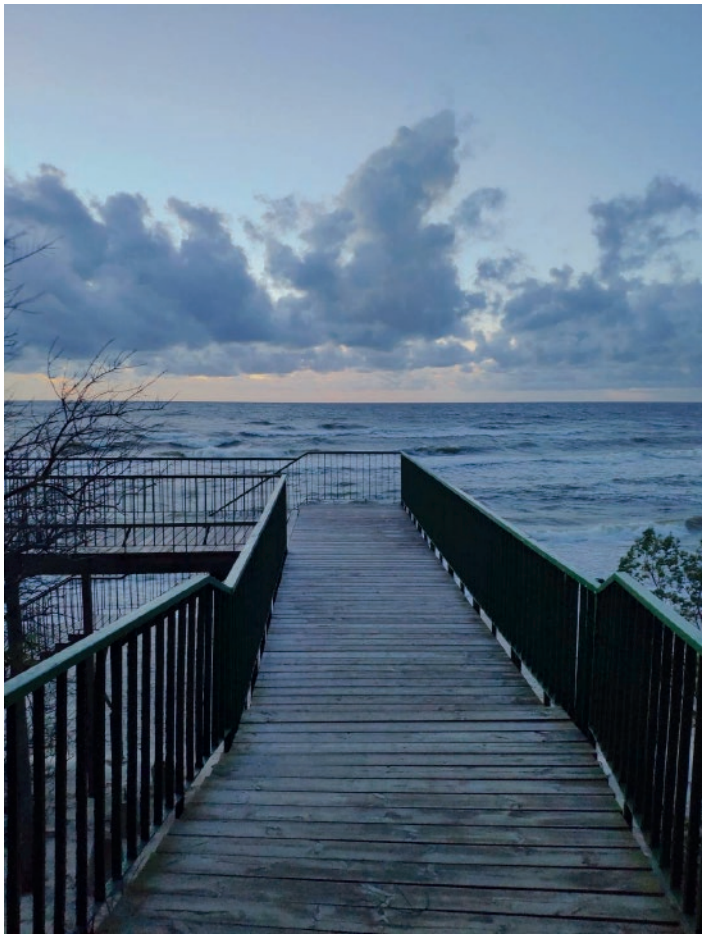
Listing 5. Zapisanie wyniki do bufora wyjściowego

```
rgb[0] = std::clamp(rgb[0], 0, 255);
rgb[1] = std::clamp(rgb[1], 0, 255);
rgb[2] = std::clamp(rgb[2], 0, 255);
*out = (uint8_t)rgb[0] | ((uint8_t)rgb[1] << 8) |
       ((uint8_t)rgb[2] << 16) | (0xff << 24);
```

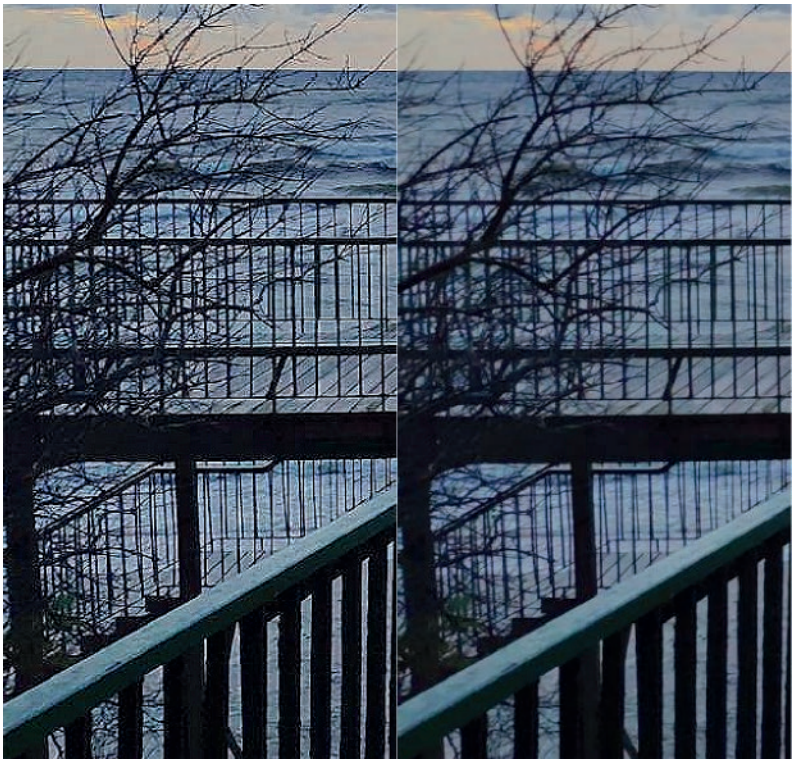
IMPLEMENTACJA WEKTOROWA

Wersja wykorzystująca SIMDy bazuje na implementacji skalarnej, jednak ze względu na konieczne modyfikacje poprzedni opis działania do niej nie przystaje.

Pracę rozpoczynamy od załadowania całych ciągów pikseli do rejestrów. Do dyspozycji mamy typy danych przechowujące liczbę bitów będącą potęgą liczby 2, dlatego – chociaż tak naprawdę interesują nas tylko 3 piksele (96 bitów) – musimy załadować więcej danych i zapeł-



Rysunek 5. Obraz oryginalny



Rysunek 6. Porównanie oryginalnego fragmentu obrazu z fragmentem po zastosowaniu filtru wyostrzającego

Rysunek 7. Obraz po zastosowaniu filtru Sobela

nić rejestr o wielkości 128 bit. Po załadowaniu każdego wiersza musimy odpowiednio przesunąć wskaźnik do tablicy danych wejściowych.

Listing 6. Ładowanie danych do rejestrów wektorowych

```
uint32_t* tmp = curr;
__m128i d0 = _mm_loadu_si128(((__m128i*)tmp));
tmp += w;
__m128i d1 = _mm_loadu_si128(((__m128i*)tmp));
tmp += w;
__m128i d2 = _mm_loadu_si128(((__m128i*)tmp));
```

Następnym krokiem jest odpowiednie przetasowanie 8-bitowych liczb w ramach 128-bitowego rejestru, przy jednoczesnym wyzerowaniu wartości 4-tego, nadmiarowego elementu. Robimy to po to, aby później ułatwić sobie dodawanie wartości poszczególnych kanałów.

Listing 7. Przetasowanie rejestru

```
__m128i smask = _mm_set_epi8(128, 11, 7, 3, 128, 10, 6, 2,
                             128, 9, 5, 1, 128, 8, 4, 0);
__m128i d0s = _mm_shuffle_epi8(d0, smask);
__m128i d1s = _mm_shuffle_epi8(d1, smask);
__m128i d2s = _mm_shuffle_epi8(d2, smask);
```

Kolejnym krokiem jest konwersja danych z 8 bit na 16 bit oraz przemnożenie ich przez odpowiednie wartości filtra. Konwersji dokonujemy po to, aby wyniki pośrednie nie wyszły poza zakres liczb 8-bitowych.

Listing 8. Mnożenie pikseli przez odpowiednie składowe filtra

```
__m256i r0_16 = _mm256_mullo_epi16(
                      _mm256_cvtepu8_epi16(d0s), f1r0);
__m256i r1_16 = _mm256_mullo_epi16(
                      _mm256_cvtepu8_epi16(d1s),  f1r1);
__m256i r2_16 = _mm256_mullo_epi16(
                      _mm256_cvtepu8_epi16(d2s), f1r2);
```

Przemnożone dane sumujemy wierszami, a następnie przeprowadzamy dodawanie lane'ów – czyli sąsiadujących ze sobą liczb, w tym wypadku 16-bitowych, w ramach tego samego rejestru. Interesuje nas tylko połowa rejestru, zawierająca faktyczne dane, dlatego za pomocą odpowiedniej operacji ekstrahujemy je do rejestru 128-bitowego. Tutaj ponownie sumujemy lane'y, dzięki czemu otrzymujemy wartości poszczególnych kanałów w odpowiednich liczbach 16-bitowych umieszczonych w 128-bitowym rejestrze.

Listing 9. Sumowanie wartości dla poszczególnych kanałów

```
__m256i rowsum = _mm256_add_epi16(
                 _mm256_add_epi16(r0_16, r1_16),
                 r2_16);
__m256i hz = _mm256_hadd_epi16(rowsum,
             _mm256_set1_epi32(0));
__m128i hz128 = _mm256_extracti128_si256(hz, 0);
__m128i hz2 = _mm_hadd_epi16(hz128,
              _mm256_extractf128_si256(hz, 1));
```

Pozostaje nam jeszcze obcięcie wartości do zakresu 0-255 oraz wybranie określonych liczb 8-bitowych z rejestru i zapisanie ich do bufora danych wyjściowych.

Listing 8. Obcięcie i zapis końcowej wartości piksela

```
__m128i clamped = _mm_max_epi16(_mm_min_epi16(hz2,
                  _mm_set1_epi16(255)),
                  _mm_set1_epi16(0));
__m128i clampedsh = _mm_shuffle_epi8(clamped,
                    _mm_set_epi8(128, 128, 128, 128,
                                 128, 128, 128, 128,
                                 128, 128, 128, 128,
                                 128, 8, 2, 0));
int rgb = _mm_extract_epi32(clampedsh, 0);
*out = rgb | ( 0xff << 24);
```

WADY I ZALETY

Ilość kodu dla obydwu wersji jest porównywalna, ale bywa, że implementacja wektorowa jest obszerniejsza niż skalarna. Problemem jest również fakt, że kod wektorowy jest niezrozumiały dla niewprawionego oka – mnemoniki SIMDowe mogą jawić się niczym losowe ciągi znaków. Ogólna czytelność kodu spada jeszcze bardziej, kiedy chcemy pokryć większą liczbę zestawów instrukcji. Sam proces adaptacji wersji skalarnej do wektorowej jest żmudny, wymaga częstego przeszukiwania dokumentacji czy rozrysowywania niektórych operacji na papierze (a przynajmniej autor wspierał się w ten sposób). Wymienione wady przeciwważy imponujący wzrost wydajności. Zarówno implementacja skalarna, jak i wektorowa nie były szczególnie zoptymalizowane – między innymi po to, aby nie pogarszać czytelności. Porównując jednak obie wersje, widzimy uderzającą różnicę w czasie potrzebnym do wykonania obliczeń. Dla obrazu o rozdzielczości 1600x1200 pikseli jest to odpowiednio 1700,85 milisekundy dla wersji skalarnej i 71,92 milisekundy dla wersji wektorowej. Oznacza to ponad 23-krotne przyspieszenie.

PODSUMOWANIE

Powyższy tekst przybliża czytelnikowi zagadnienie wykorzystania operacji SIMD udostępnianych przez procesory w architekturze x86 oraz ARM. Przedstawiono kroki niezbędne do użycia funkcji intrinsic i zaprezentowano przykładowe zadanie, dla którego korzyści ze stosowania SIMD są wyraźne i jednoznaczne. Ich osiągnięcie wymaga jednak określonego nakładu pracy, a zakres zastosowań jest ograniczony. Sięganie po tego typu rozwiązania jest uzasadnione wtedy, kiedy wykonujemy te same operacje na wielu danych jednocześnie i robimy to wielokrotnie. Wówczas wrzucamy szósty bieg naszego procesora.

Kod programu testowego umieszczono pod adresem:

» https://github.com/m1awicki/simd-demo.

W sieci

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
https://developer.arm.com/architectures/instruction-sets/intrinsics/
https://en.wikipedia.org/wiki/ILLIAC_IV

MARCIN ŁAWICKI

Autor zajmuje się programowaniem od kilkunastu lat. W tym czasie miał okazję pracować nad różnego rodzaju oprogramowaniem, począwszy od gier, przez aplikacje dla sektora bankowego i telekomunikacji, do aplikacji mobilnych, desktopowych czy IoT. Aktualnie pracuje jako manager zespołu programistów w Huuuge Games, gdzie pomaga tworzyć i rozwijać technologie własne firmy.