Ministerstwo Nauki
i Szkolnictwa Wyższego

Regionalna Inicjatywa Doskonałości w Dyscyplinach
Informatyki, Elektrotechniki, Elektroniki, Automatyki i Robotyki
na Politechnice Częstochowskiej

Projekt w ramach programu Regionalna Inicjatywa Doskonałości, decyzja nr 020/RID/2018/19

Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną

Dr hab. inż. Łukasz Szustak
lszustak@icis.pcz.pl

Dr inż. Kamil Halbiniak
khalbniak@icis.pcz.pl

Politechnika Częstochowska

Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną

1. Wprowadzenie do współczesnych architektur komputerowych oraz programowania równoległego
2. Przegląd modeli i standardów programowania równoległego z pamięcią współdzieloną
3. Środowisko OpenMP programowania maszyn z pamięcią wspólna, w tym procesorów wielordzeniowych
4. Zrównoleglanie pętli oraz równoważenie obciążenia w systemach równoległych z pamięcią współdzieloną
5. Zrównoleglanie programów z wykorzystaniem modelu typu data parallelism
6. Zrównoleglanie programów z wykorzystaniem modelu typu task parallelism
7. Mapowanie obliczeń w hybrydowych systemach komputerowych z akceleratorami
8. Przykłady zrównoleglania obliczeń numerycznych

Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną

1. Wprowadzenie do współczesnych architektur komputerowych oraz programowania równoległego
2. Przegląd modeli i standardów programowania równoległego z pamięcią współdzieloną
3. Środowisko OpenMP programowania maszyn z pamięcią wspólna, w tym procesorów wielordzeniowych
4. Zrównoleglanie pętli oraz równoważenie obciążenia w systemach równoległych z pamięcią współdzieloną
5. **Zrównoleglanie programów z wykorzystaniem modelu typu data parallelism**
6. Zrównoleglanie programów z wykorzystaniem modelu typu task parallelism
7. Mapowanie obliczeń w hybrydowych systemach komputerowych z akceleratorami
8. Przykłady zrównoleglania obliczeń numerycznych

Temat nr 5

Zrównoleglanie programów z wykorzystaniem modelu typu data parallelism

Dr hab. inż. Łukasz Szustak
lszustak@icis.pcz.pl

Politechnika Częstochowska

Plan prezentacji

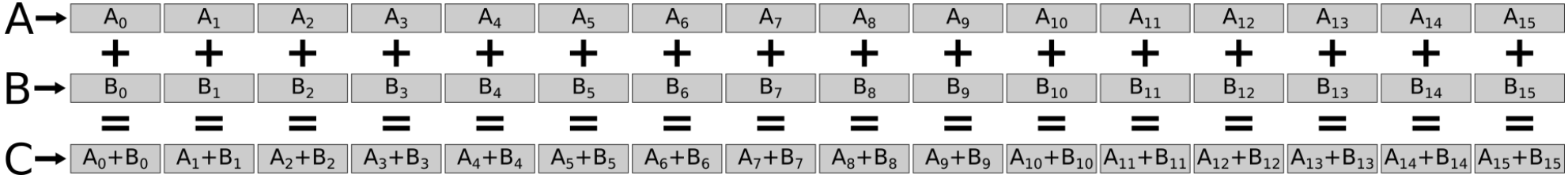
- Koncepcja modelu data parallelism na przykładzie wektoryzacji obliczeń
- Automatyczna wektoryzacja obliczeń
- Półautomatyczna wektoryzacja obliczeń
- Manualne wektoryzacja obliczeń

Koncepcja modelu data parallelism

```
for(int i=0; i<16; ++i) {
    C[i]=A[i]+B[i];
}
```

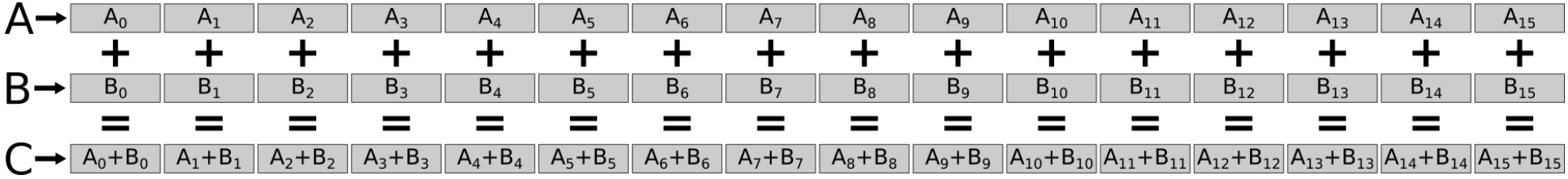
Koncepcja modelu data parallelism

```
for(int i=0; i<16; ++i) {
    C[i]=A[i]+B[i];
}
```



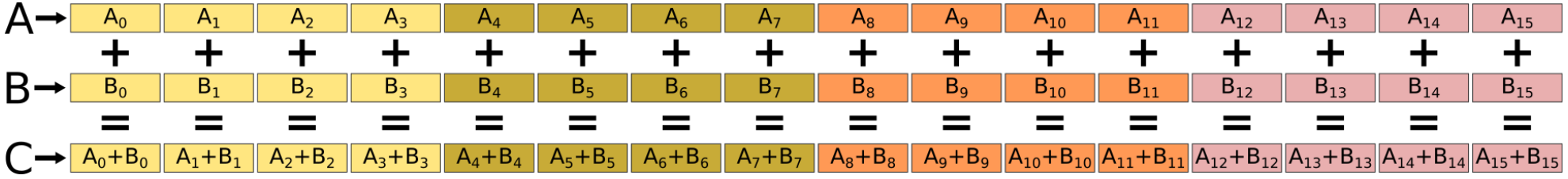
Koncepcja modelu data parallelism

```
#pragma omp for schedule(static)
for(int i=0; i<16; ++i) {
    C[i]=A[i]+B[i];
}
```



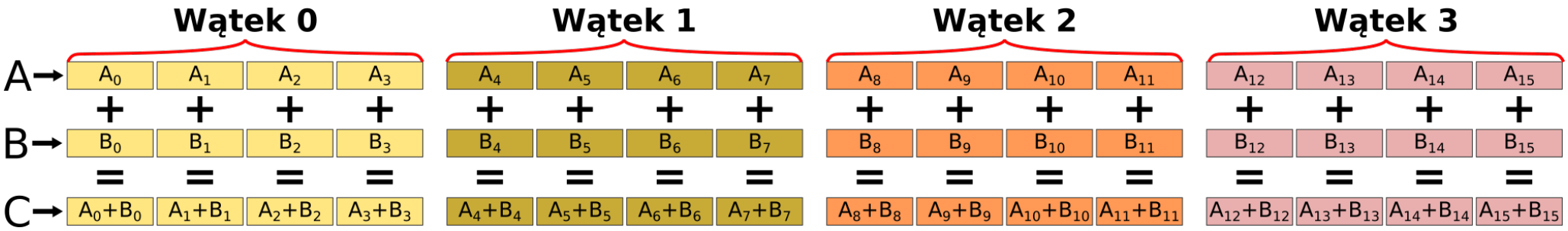
Koncepcja modelu data parallelism

```
#pragma omp for schedule(static)
for(int i=0; i<16; ++i) {
    C[i]=A[i]+B[i];
}
```



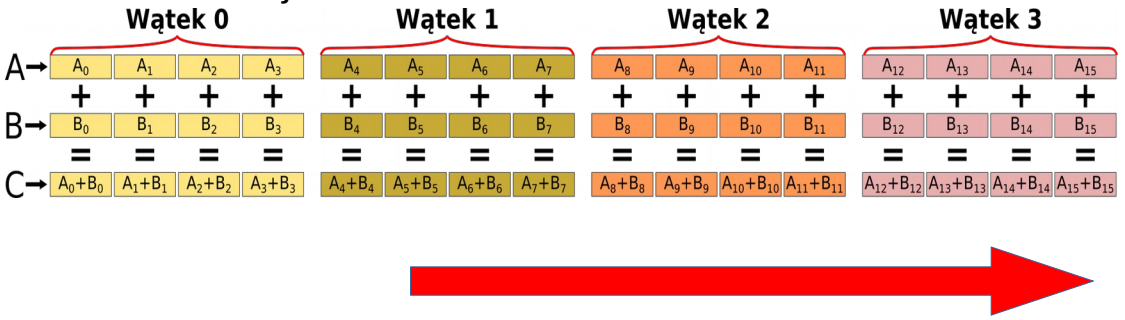
Koncepcja modelu data parallelism

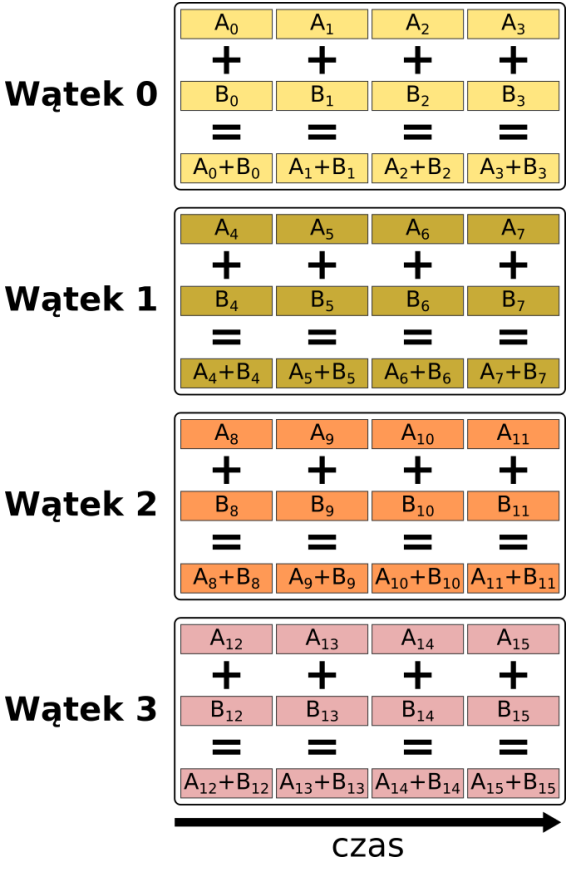
```
#pragma omp for schedule(static)
for(int i=0; i<16; ++i) {
    C[i]=A[i]+B[i];
}
```



Koncepcja modelu data parallelism

```
#pragma omp for schedule(static)
for(int i=0; i<16; ++i) {
    C[i]=A[i]+B[i];
}
```





Koncepcja modelu data parallelism

Koncepcja modelu data parallelism

SIMD

(**S**ingle **I**nstruction, **M**ultiple **D**ata)

Architektura SIMD

- SIMD (Single Instruction, Multiple Data) - przetwarzanych jest wiele danych przez jeden wykonywany program - tzw. komputery wektorowe

A vector processor or array processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to the scalar processors, whose instructions operate on single data items.

Architektura SIMD

- SIMD (Single Instruction, Multiple Data) - przetwarzanych jest wiele danych przez jeden wykonywany program - tzw. komputery wektorowe
- Cray 1 (1976) :
  - 8 MB RAM
  - 160 MFLOPS
  - chłodzenie sprężarkowe
  - charakterystyczny kształt litery „C”

https://www.cray.com/blog/original-cray-1-supercomputer-unveiled-seattle/

Architektura SIMD

- Współczesne architektury procesorów różnych producentów wspierają wektoryzację, np..:
  - **x86**:

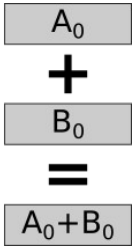
    SSE 128 bit, AVX(2) 256 bit, AVX-512 512 bit
  - **ARM**:

    NEON 128 bit
  - **POWER:**

    AltiVec/VMX/VSX 128 bit, QPX 256 bit
  - **SPARC**:
    HPC-ACE 128 bit, HPC-ACE2 256 bit

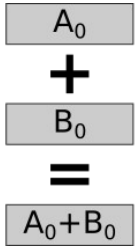
Architektura SIMD

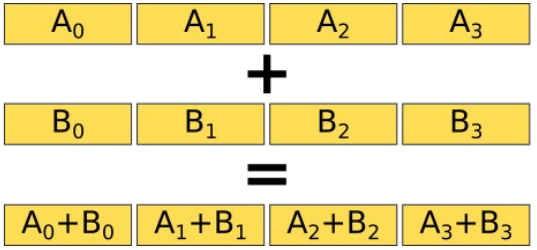
Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji



Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji

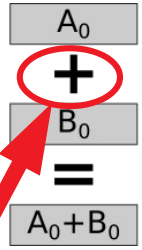




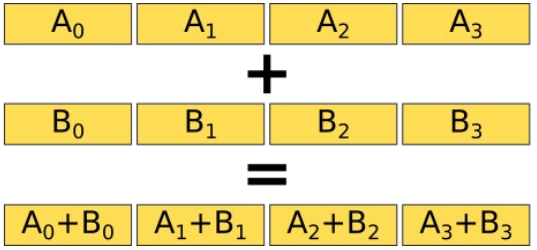
Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji



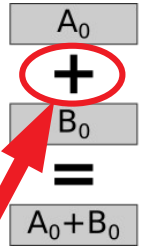
Skalarna instrukcja dodawania

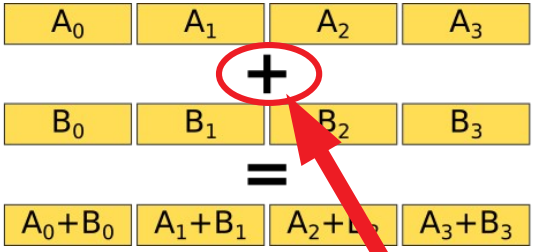


Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji





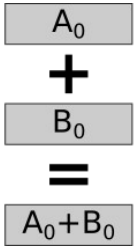
Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

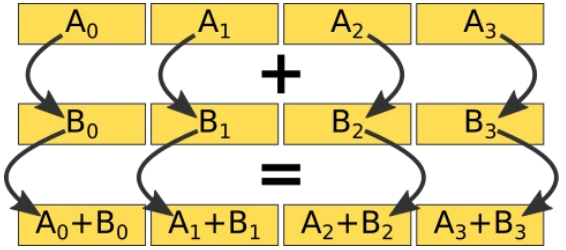
Skalarna instrukcja dodawania

Wektorowa instrukcja dodawania

Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji

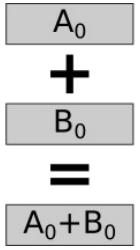


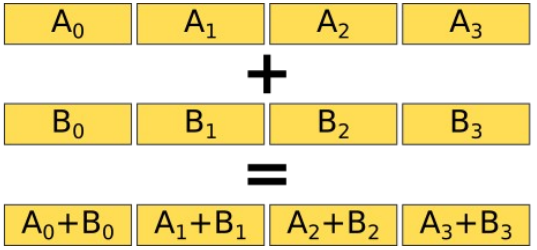


Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji



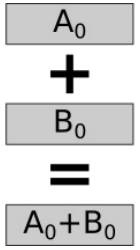


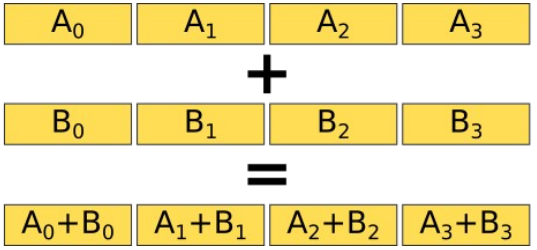
Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

W powyższym przypadku przetwarzanych jest
4x więcej elementów

Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji





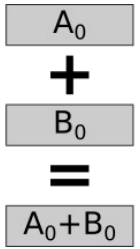
Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

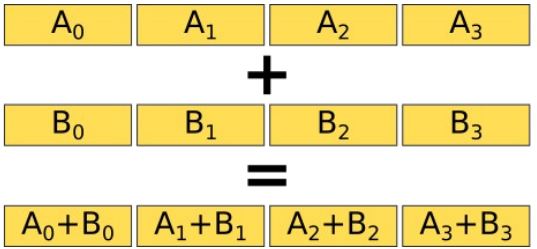
W powyższym przypadku przetwarzanych jest 4x więcej elementów

Zatem, potencjalny zysk wynika z liczby elementów przetwarzanych równocześnie

Architektura SIMD

Obliczenia skalarne
Pojedynczy element przetwarzany z pomocą pojedynczej operacji





Obliczenia wektorowe
Zbiór elementów (wektor) przetwarzany z pomocą pojedynczej operacji

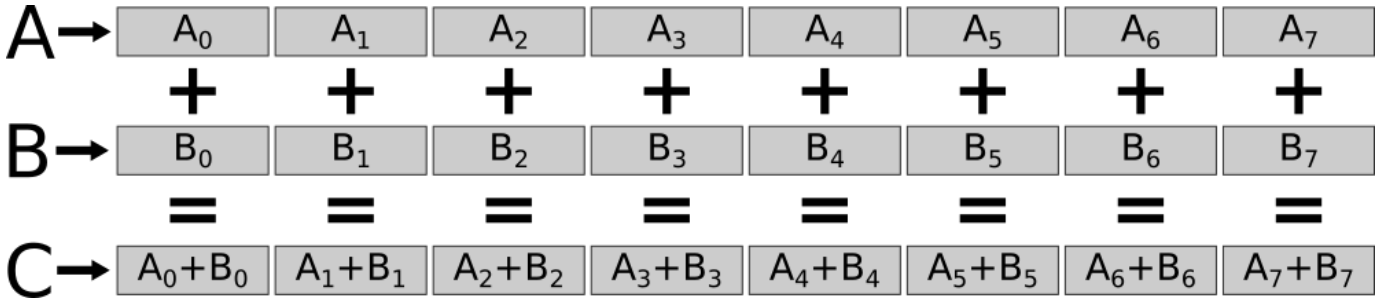
W powyższym przypadku przetwarzanych jest 4x więcej elementów

Zatem, potencjalny zysk wynika z liczby elementów przetwarzanych równocześnie

Liczba elementów zależy od typu danych oraz rozmiaru rejestrów

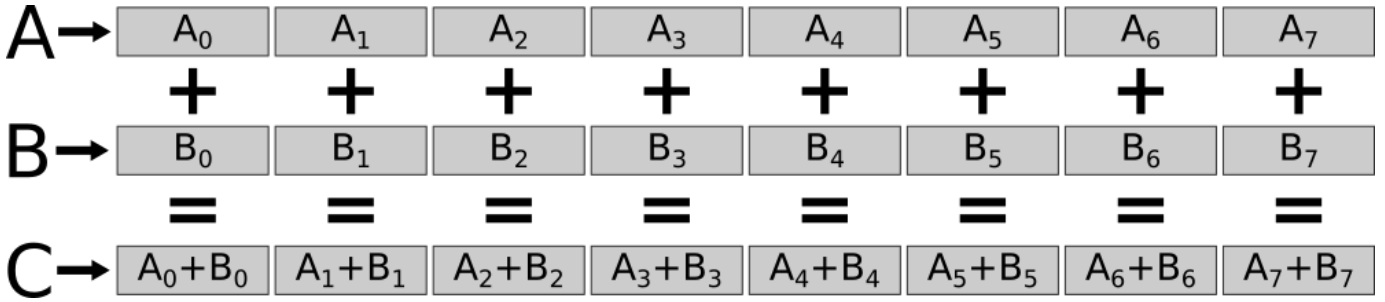
Architektura SIMD

Obliczenia Skalarne



Architektura SIMD

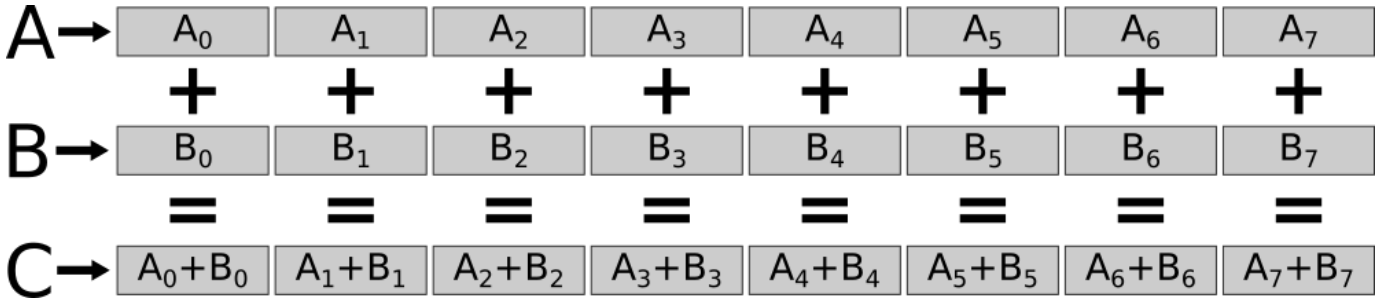
Obliczenia Skalarne



8 operacji
dodawania
z użyciem
8 pojedynczych
elementów tablicy

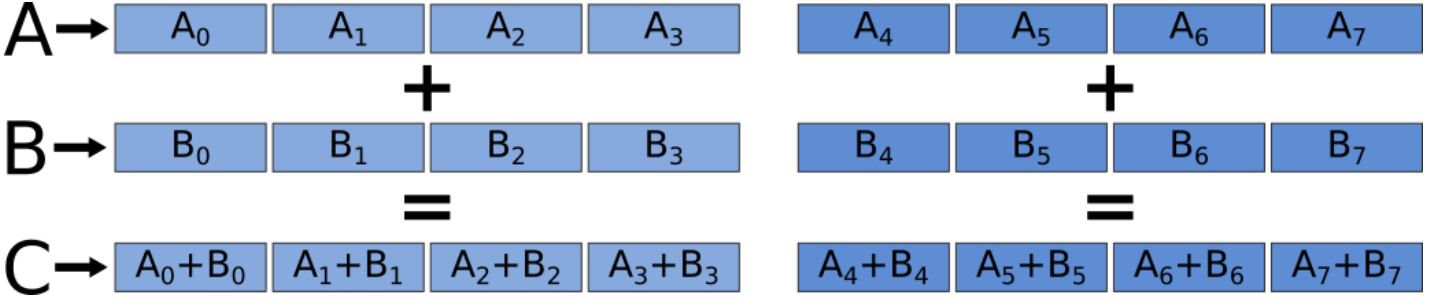
Architektura SIMD

Obliczenia Skalarne



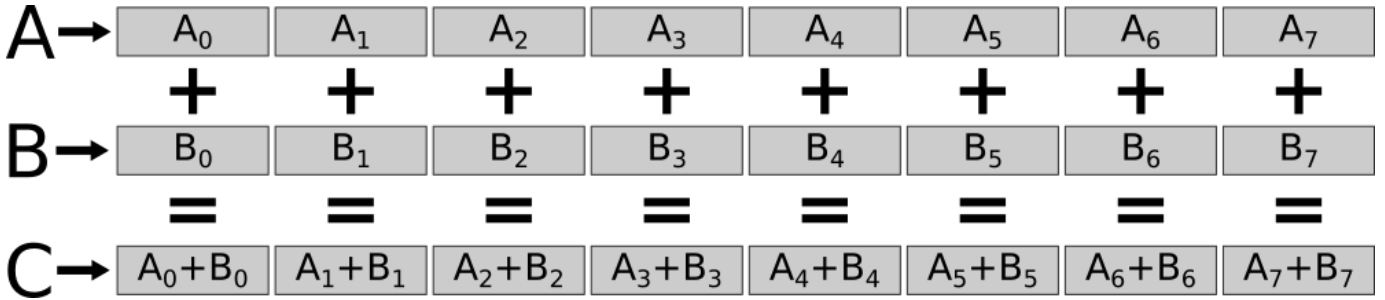
8 operacji dodawania z użyciem 8 pojedynczych elementów tablicy

Obliczenia Wektorowe



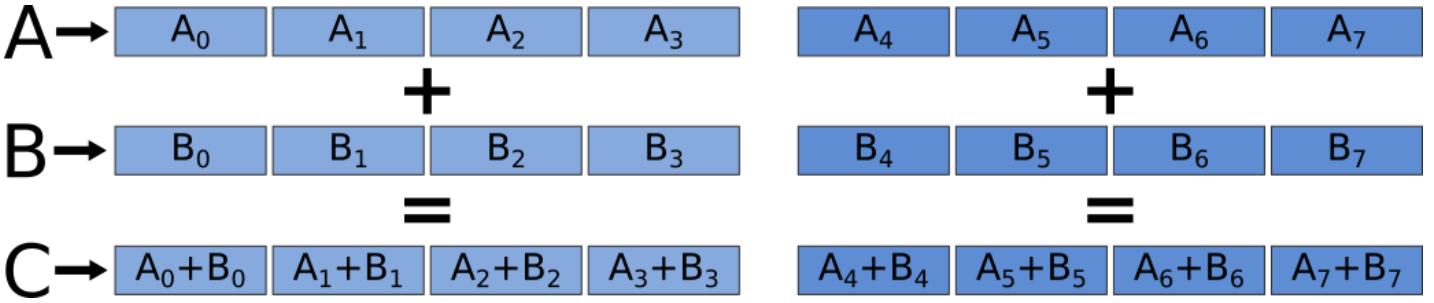
Architektura SIMD

Obliczenia Skalarne



8 operacji
dodawania
z użyciem
8 pojedynczych
elementów tablicy

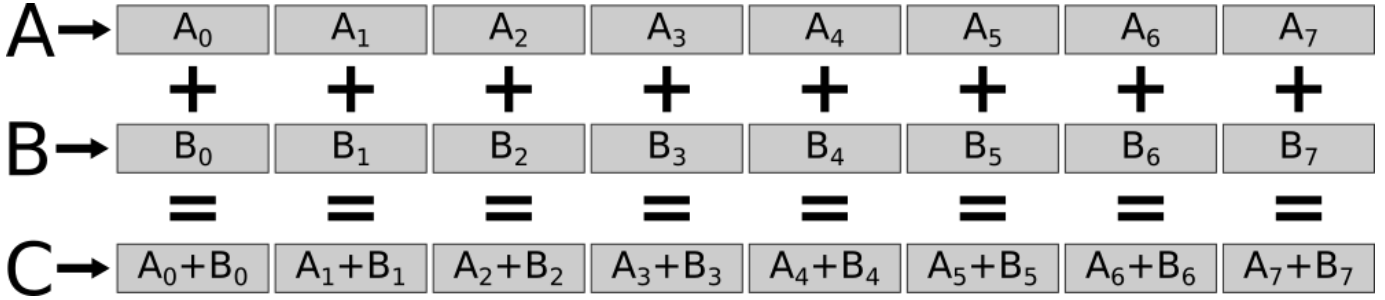
Obliczenia Wektorowe



2 operacje
dodawania
z użyciem 2
czteroelementowych
wektorów

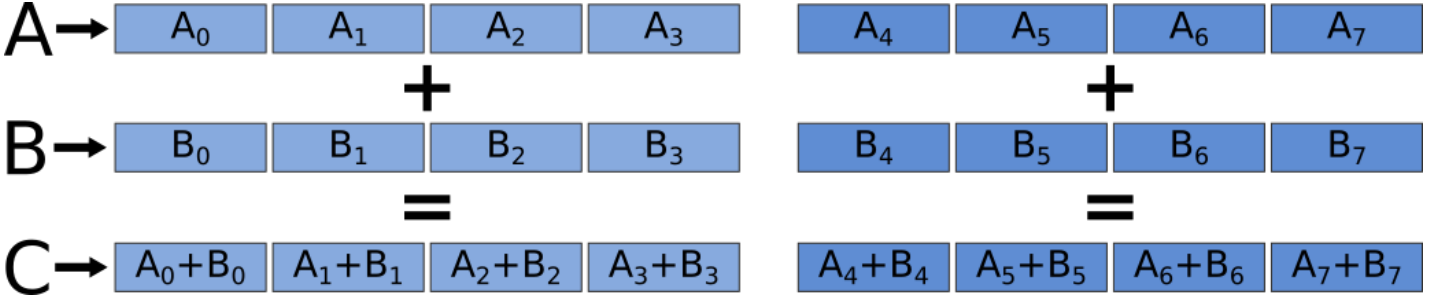
Architektura SIMD

Obliczenia Skalarne



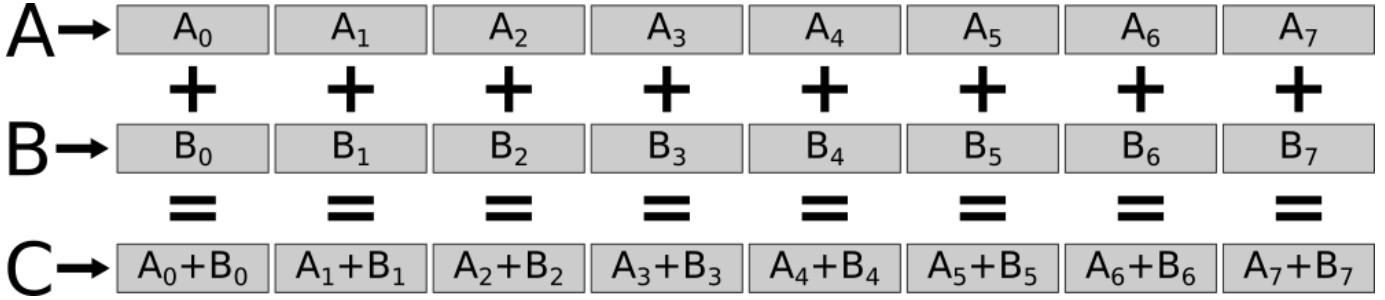
Zakładając, że operacja dodawania zajmuje 1 cykl w obu przypadkach

Obliczenia Wektorowe

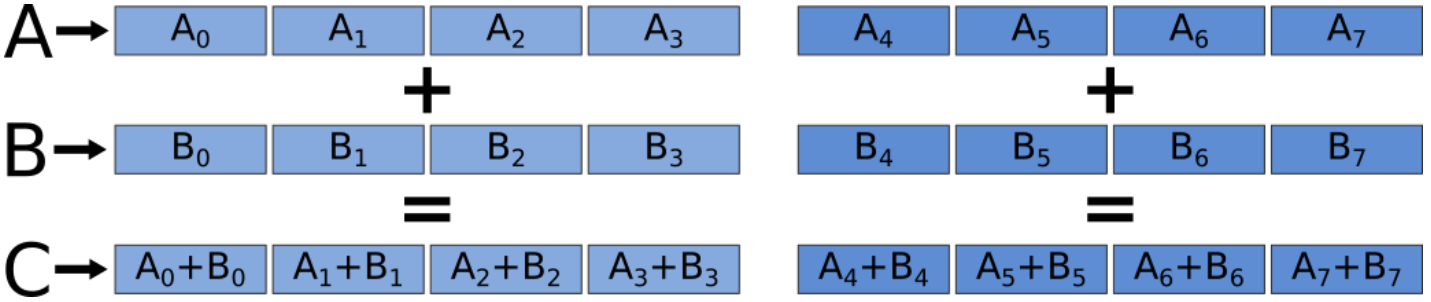


Architektura SIMD

Obliczenia Skalarne



Obliczenia Wektorowe



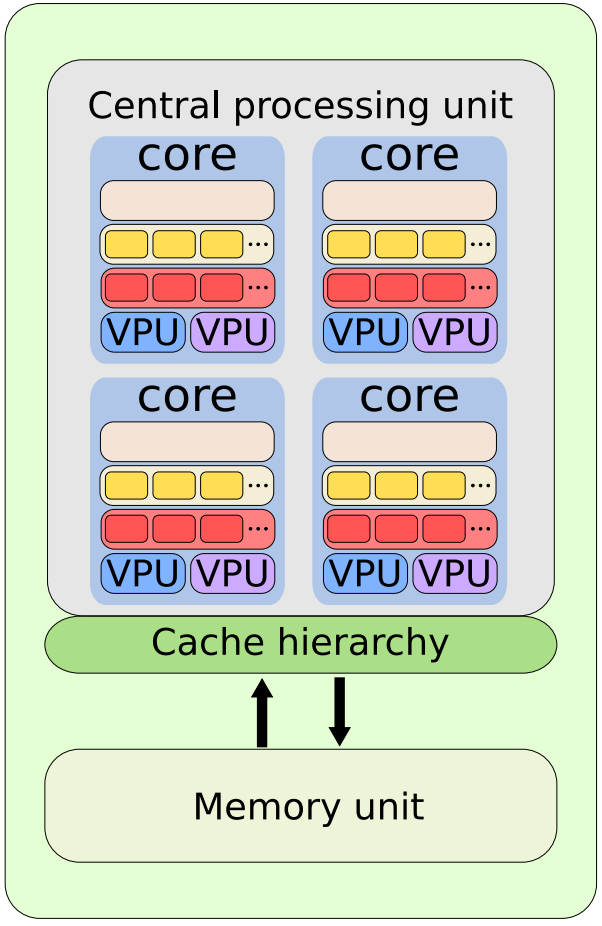
Zakładając, że operacja dodawania zajmuje 1 cykl w obu przypadkach

Użycie architektury SIMD umożliwi redukcję liczby cykli z 8 do 2:

4x mniej cykli

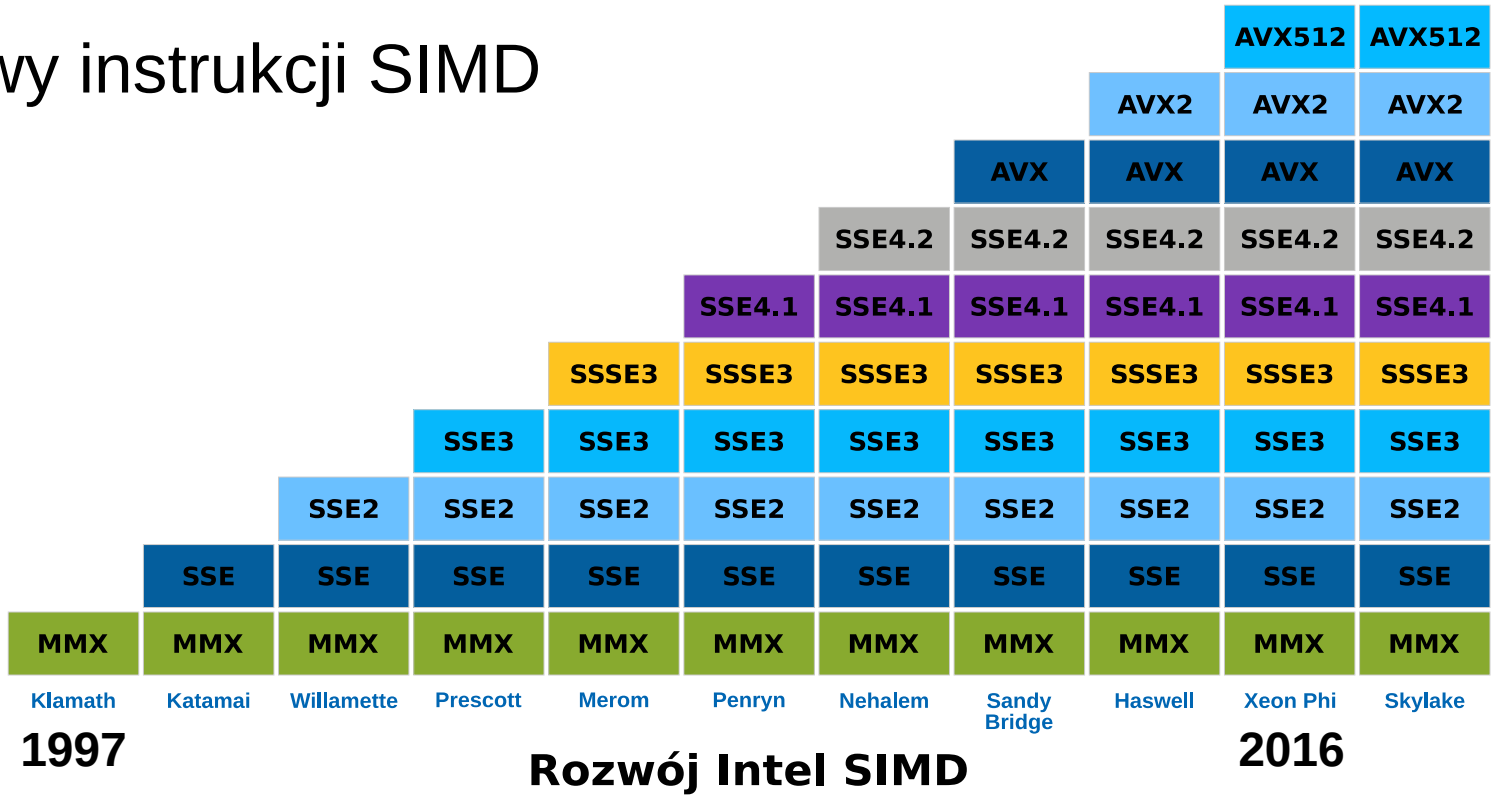
Architektura SIMD

- Obliczenia wektorowe realizowane są z pomocą jednostek wektorowych **VPU**
- W każdym rdzeniu procesora wspierającego SIMD znajduję się co najmniej jedna jednostka VPU
- Każdy rdzeń zawiera zbiór rejestrów dedykowanych do wykonywania instrukcji wektorowe
- Rozmiar rejestrów wektorowych zależy od danej architektury procesora
- Każda jednostka VPU obsługuję zestaw instrukcji (np.. +,-,*,/) dedykowany dla danej architektury procesora



Rozwój architektury SIMD na przykładzie procesorów firmy Intel

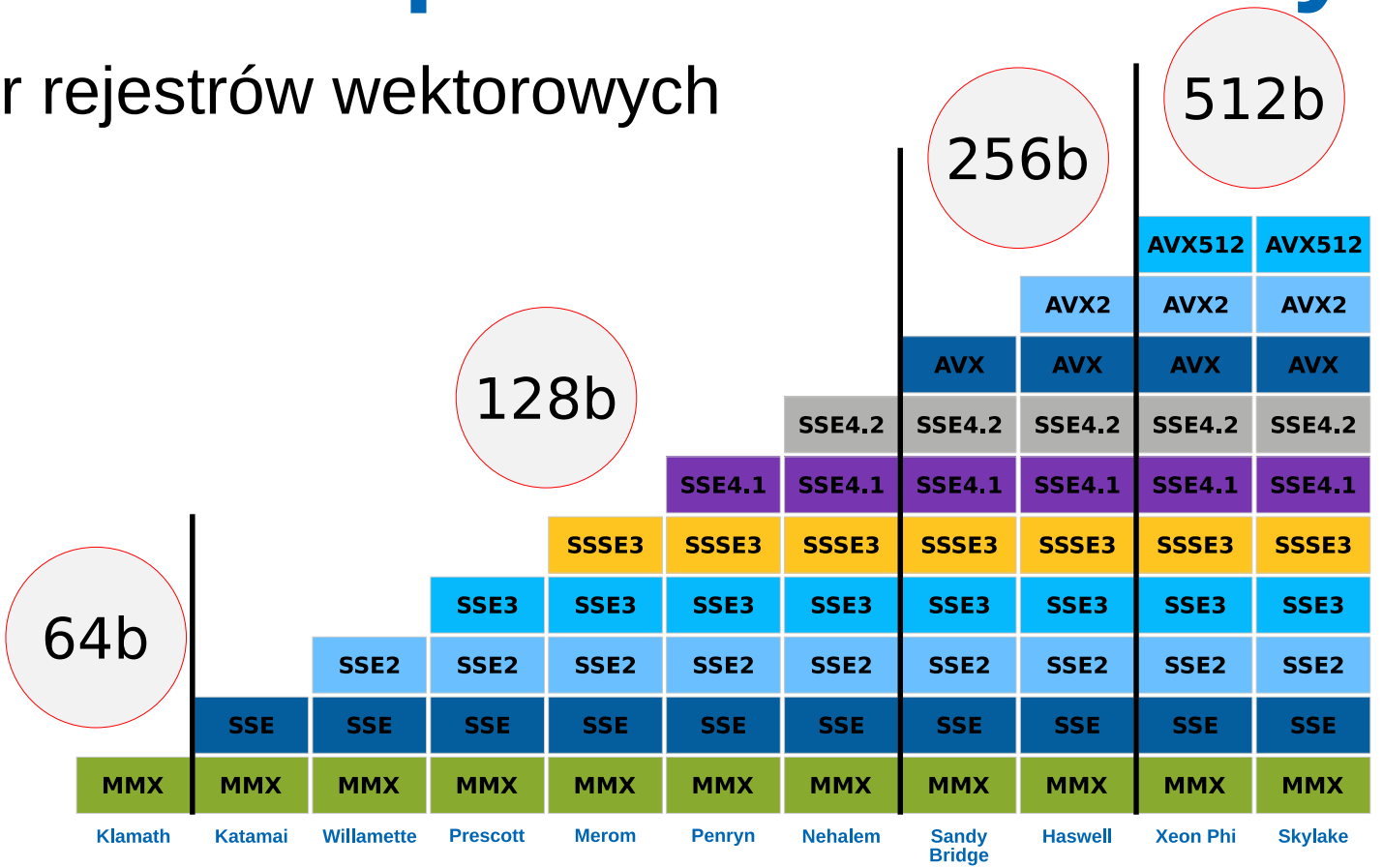
- Zestawy instrukcji SIMD



https://software.intel.com/sites/landingpage/IntrinsicsGuide/

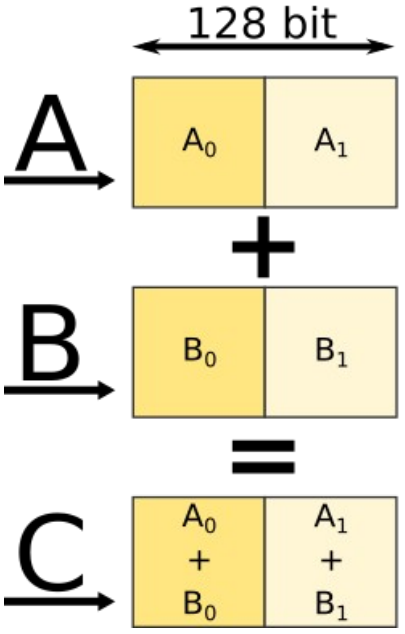
Rozwój architektury SIMD na przykładzie procesorów firmy Intel

- Rozmiar rejestrów wektorowych



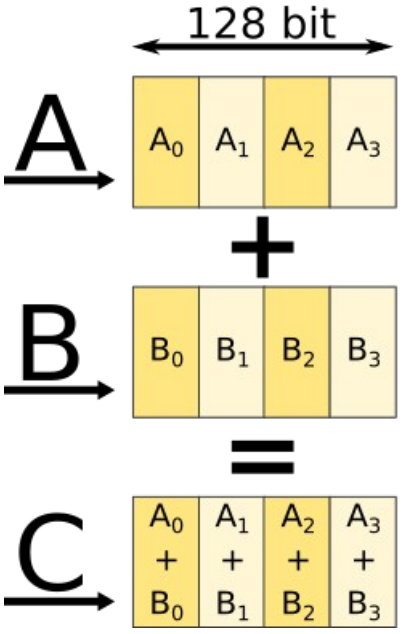
Architektura SIMD na przykładzie procesorów firmy Intel

- Zestaw instrukcji **SSE**
- **128** bitowe rejestry „XMM”
- **2 element**y typu double (64 bity)



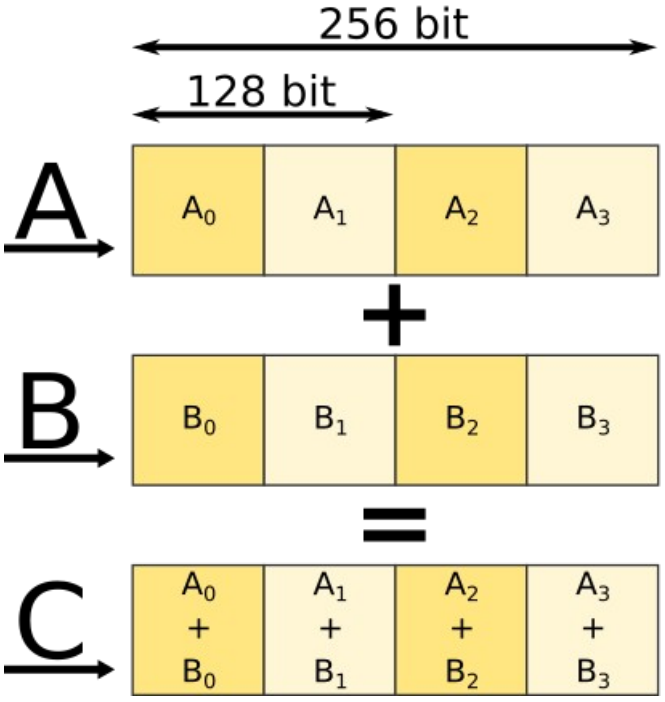
Architektura SIMD na przykładzie procesorów firmy Intel

- Zestaw instrukcji **SSE**
- **128** bitowe rejestry „XMM”
- **4 elementy** typu float (32 bity)



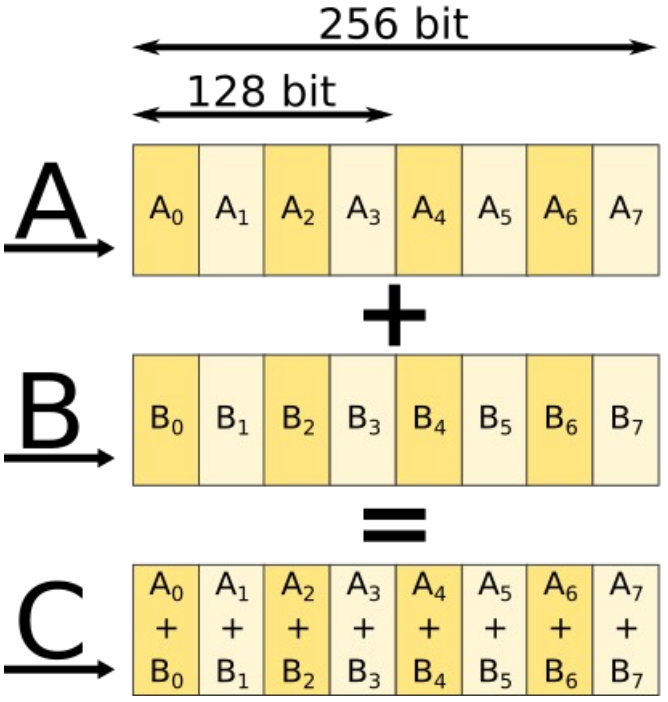
Architektura SIMD na przykładzie procesorów firmy Intel

- Zestaw instrukcji **AVX**
- **256** bitowe rejestry „YMM”
- **4 elementy** typu double (64 bity)



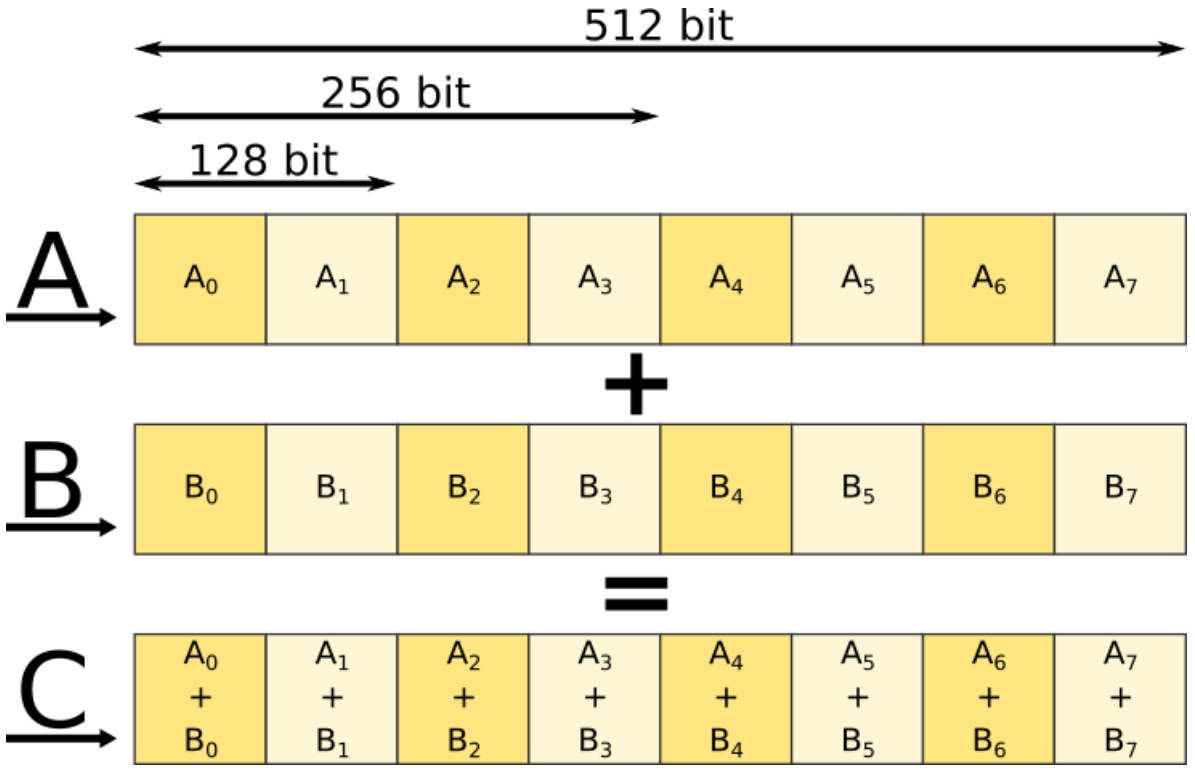
Architektura SIMD na przykładzie procesorów firmy Intel

- Zestaw instrukcji **AVX**
- **256** bitowe rejestry „YMM”
- **8 elementów** typu float (32 bity)



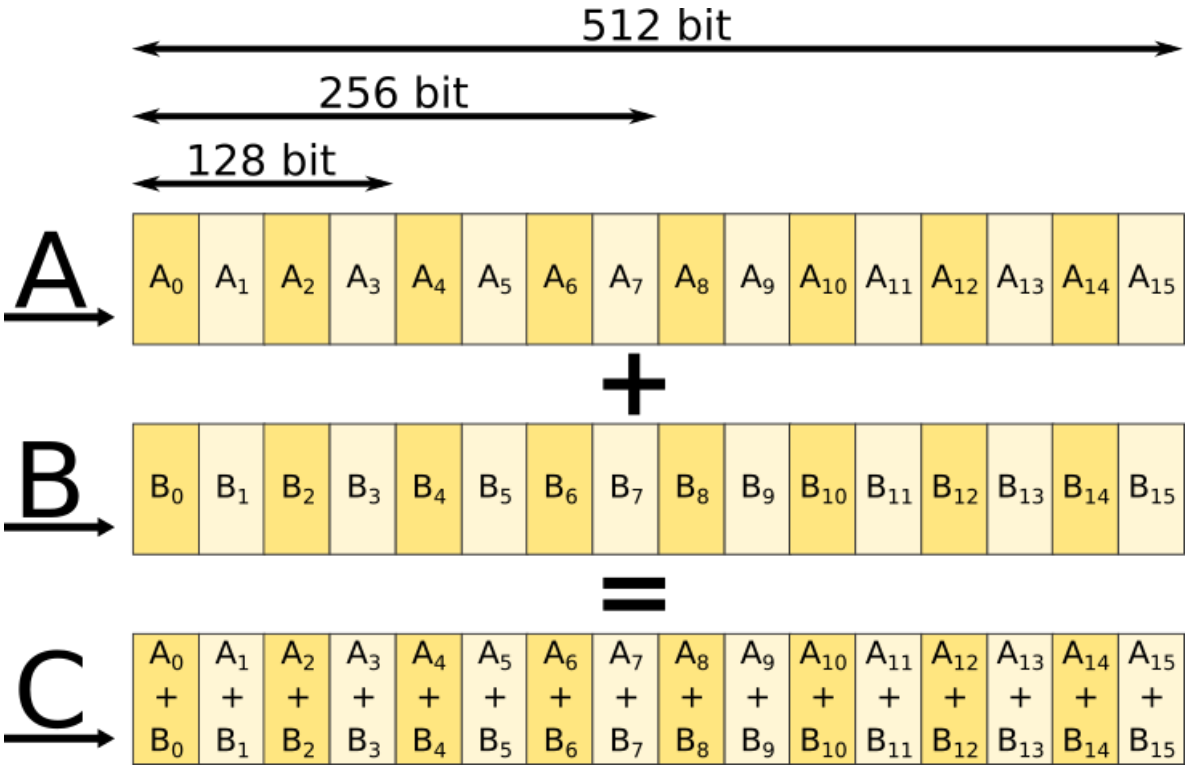
Architektura SIMD na przykładzie procesorów firmy Intel

- Zestaw instrukcji **AVX-512**
- **512** bitowe rejestry „ZMM”
- **8 elementów** typu double (64 bity)



Architektura SIMD na przykładzie procesorów firmy Intel

- Zestaw instrukcji **AVX-512**
- **512** bitowe rejestry „ZMM”
- **16 elementów** typu float (32 bity)

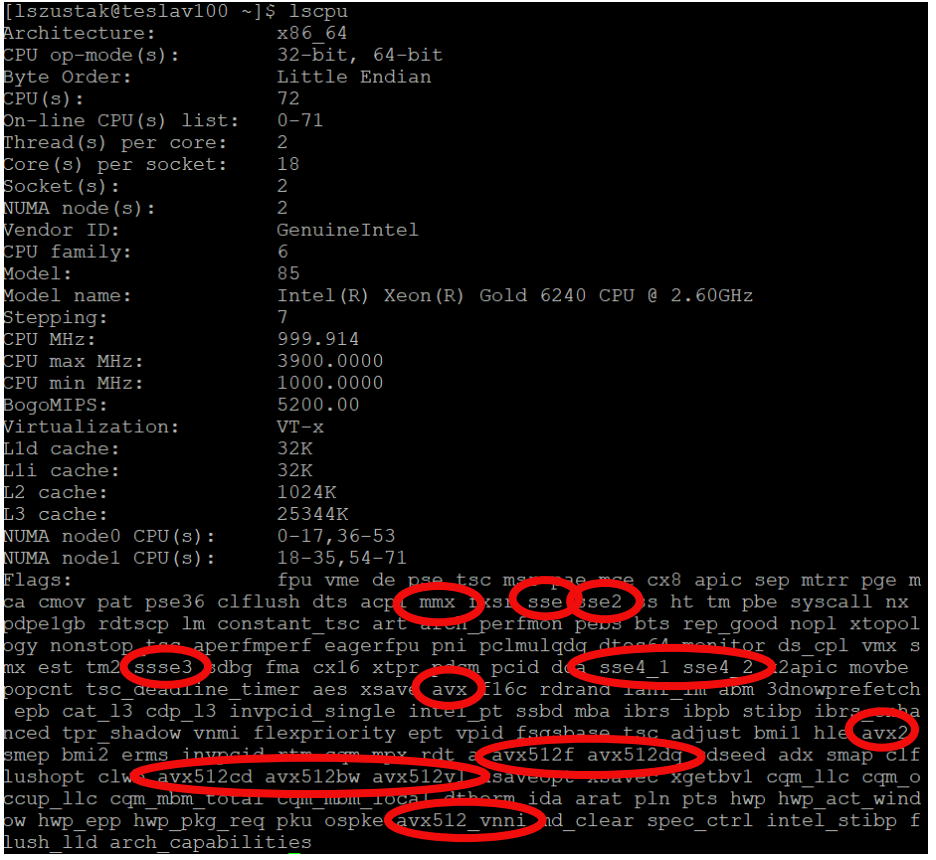


Rozwój architektury SIMD na przykładzie procesorów firmy Intel

```
rid@gpu2:~/RID/T5$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
Address sizes:           46 bits physical, 48 bits virtual
CPU(s):                  72
On-line CPU(s) list:     0-71
Thread(s) per core:      2
Core(s) per socket:      18
Socket(s):               2
NUMA node(s):            2
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   63
Model name:              Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
Stepping:                2
CPU MHz:                 1198.211
CPU max MHz:             3600,0000
CPU min MHz:             1200,0000
BogoMIPS:                4589.45
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                46080K
NUMA node0 CPU(s):       0-17,36-53
NUMA node1 CPU(s):       18-35,54-71
Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
 pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdts
cp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
 aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx1
6 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsav
e avx f16c rdrand lahf_lm abm cpuid_fault epb invpcid_single pti intel_ppin ssbd i
brs_ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bm
i1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dtherm ida arat
pln pts md_clear flush_l1d
```

- Różne architektury CPU wspierają różne zestawy instrukcji, np..
  Intel Xeon E5-2699v3

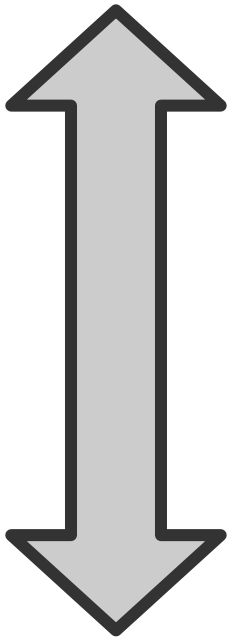
Rozwój architektury SIMD na przykładzie procesorów firmy Intel



- Różne architektury CPU wspierają różne zestawy instrukcji, np..
  Intel Xeon Gold 6240

Wektoryzacja obliczeń

Łatwe w użyciu



Duża kontrola

Biblioteki
(MKL, OpenBLAS, BLIS, tw, numpy, OpenCV)

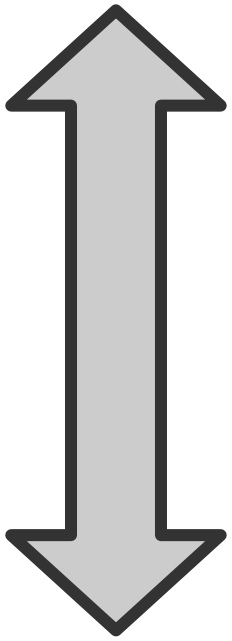
Automatyczna wektoryzacja obliczeń
(kompilatory, np.. gcc, intel icc)

Półautomatyczna wektoryzacja obliczeń
(kompilatory + odpowiednie przygotowanie kodu)

Manualna wektoryzacja obliczeń
(instrukcje wbudowane, np.. intrinsics, asembler)

Wektoryzacja obliczeń

Łatwe w użyciu



Duża kontrola

Biblioteki
(MKL, OpenBLAS, BLIS, tw, numpy, OpenCV)

Automatyczna wektoryzacja obliczeń
(kompilatory, np.. gcc, intel icc)

Półautomatyczna wektoryzacja obliczeń
(kompilatory + odpowiednie przygotowanie kodu)

Manualna wektoryzacja obliczeń
(instrukcje wbudowane, np.. intrinsics, asembler)

Automatyczna wektoryzacja obliczeń

Automatyczna wektoryzacja obliczeń

- Automatyczna wektoryzacja realizowana jest przez kompilator
- Kompilator analizuję kod i znajduję miejsce w programie (najczęściej pętle), które mogą zostać zwektoryzowane
- Proces automatycznej wektoryzacji jest przenośny między różnymi architekturami – jedynym wymogiem jest ponowna rekompilacja programu

Automatyczna wektoryzacja obliczeń

- Proces automatycznej kompilacji wymga włącznie odpowiedniej flagi:
  - Kompilatory firmy Intel: wymagana jest flaga optymalizacji na minimalnym poziomie -O2
  - Kompilatory GNU: wymagana jest flaga optymalizacji -O3 lub -ftree-vectorize
- Użytkownik może wyłączyć proces automatycznej wektoryzacji wykorzystując flagę -no-vec -no simd w przypadku kompilatorów firmy Intel lub -fno-tree-vectorize w przypadku kompilatorów gcc

Automatyczna wektoryzacja obliczeń

- Aby uzyskać proces automatycznej wektoryzacji jak najbardziej dopasowany do danej architektury trzeba dodać flagi dedykowane do konkretnej architektury

Architektura SIMD	Kompilator Intel	Kompilator GNU
Procesory wspierające AVX512 (512 bity)	-xCommon-AVX512 -xCore-AVX512 -qopt-zmm-usage=high -march=native	-mfma -mavx512f -mavx512cd -march=native
Procesory wspierające AVX (256 bity)	-mavx -march=native	-mavx -march=native
Procesory wspierające SSE (128 bity)	-msse -march=native	-msse -march=native

Automatyczna wektoryzacja obliczeń

- Zaletą kompilatorów jest możliwość również generowania raportów diagnostycznych dotyczących automatycznej wektoryzacji
- Wygenerowanie raportu dotyczącego wektoryzacji wymaga zastosowania odpowiednich flag w procesie wektoryzacji:
  - Kompilator GNU:

    -fopt-info-vec lub **-fopt-info-vec-optimized**: flagi informują, które pętle zostały automatycznie zwektoryzowane

    -fopt-info-vec-missed: flaga przedstawia szczegółowe informacje, które pętle nie zostały zwektoryzowane

    -fopt-info-vec-note: flaga przedstawia szczegółowe informacje o procesie automatycznej wektoryzacji poszczególnych pętli

    -fopt-info-vec-all: wszystkie poprzednie opcje razem
  - Kompilator Intel: -qopt-report=[n], gdzie n określa poziom szczegółowości raportu

Półautomatyczna wektoryzacja obliczeń

Półautomatyczna wektoryzacja

- Proces półautomatycznej wektoryzacji wymaga stosowania odpowiednich „podpowiedzi” dla kompilatora w celu efektywniejszego wykorzystania jednostek wektorowych
- Często wymagana jest również przebudowania kodu programu, która ułatwia kompilatorowi proces wektoryzacji
- Proces półautomatycznej wektoryzacji można zrealizować za pomocą
  - Dyrektyw OpenMP
  - Dyrektyw kompilatora firmy Intel

Półautomatyczna wektoryzacja

- **Dyrektyw OpenMP**,
  - Dyrektywy obsługiwane są przez kompilatory rożnych firm
  - Programista analizuje kod i informuje kompilator jak zwektoryzować kod
  - Kompilator „musi” dokonać wektoryzacji
- **Dyrektyw kompilatora firmy Intel**,
  - Dyrektywy są dedykowane dla kompilatorów Intel
  - Kompilator analizuje kod i sam podejmuje decyzję o procesie wektoryzacji

Półautomatyczna wektoryzacja na przykładzie OpenMP

- OpenMP oferuje możliwość wektoryzacji obliczeń
- Dyrektywa OpenMP SIMD zleca kompilatorowi polecenie zwektoryzwoania danej pętli
  - Kompilator ma obowiązek wektoryzji zgodnie z założeniami użytkownika, który jest odpowiedzialny za poprawność definiowanych założeń użytkownika
- Wektoryzacja pętli realizowana jest poprzez dodanie odpowiedniej adnotacji przed pętlą, której kolejno występujące po sobie iteracje zostaną przebudowane

Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)


for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)


for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

- count = 16

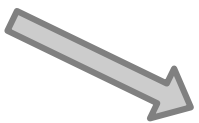
Półautomatyczna wektoryzacja na przykładzie OpenMP

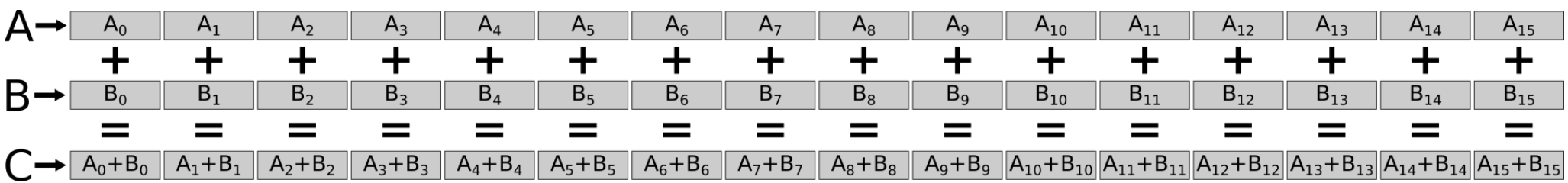
```
double a[n];
double b[n];
double c[n];

(...)


for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

- count = 16





Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

- count = 16
- #pragma omp SIMD

Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

- count = 16
- #pragma omp SIMD
- **AVX** (flaga -mavx)

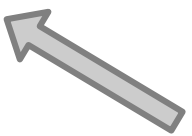
Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i+=4) {
    c[i+0] = a[i+0]+b[i+0];
    c[i+1] = a[i+1]+b[i+1];
    c[i+2] = a[i+2]+b[i+2];
    c[i+3] = a[i+3]+b[i+3];
}
```

- count = 16
- #pragma omp SIMD
- **AVX** (flaga -mavx)



Kompilator dzieli pętle na bloki o liczbie iteracji dopasowanej do rozmiaru rejestru oraz użytego typu danych:

np. dla AVX oraz typu double liczba iteracji dla pojedynczego bloku wynosi 4

Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i+=4) {
    c[i+0] = a[i+0]+b[i+0];
    c[i+1] = a[i+1]+b[i+1];
    c[i+2] = a[i+2]+b[i+2];
    c[i+3] = a[i+3]+b[i+3];
}
```

- count = 16
- #pragma omp SIMD
- **AVX** (flaga -mavx)

W ramach każdej iteracji, kompilator zamieni cztery instrukcje skalarne na jedną instrukcję wektorową: vaddpd ymm, ymm, ymm

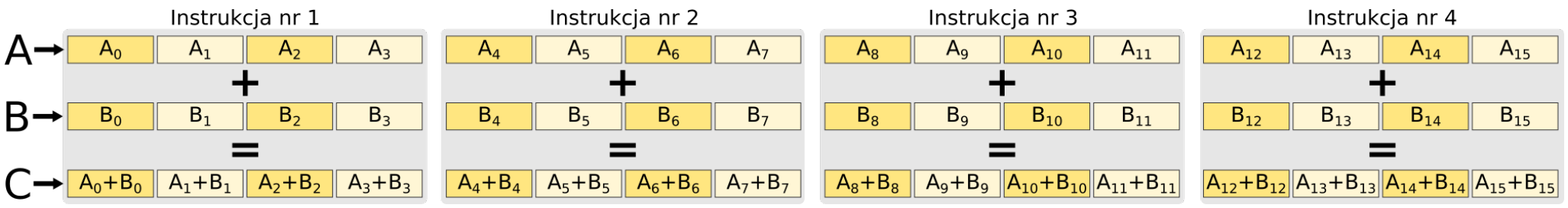
Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i+=4) {
    c[i+0] = a[i+0]+b[i+0];
    c[i+1] = a[i+1]+b[i+1];
    c[i+2] = a[i+2]+b[i+2];
    c[i+3] = a[i+3]+b[i+3];
}
```

- count = 16
- #pragma omp SIMD
- **AVX** (flaga -mavx)



Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

- count = 16
- #pragma omp SIMD

Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

- count = 16
- #pragma omp SIMD
- **AVX-512** (flagi -mavx512f -mavx512cd)

Półautomatyczna wektoryzacja na przykładzie OpenMP

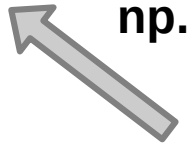
```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i+=8) {
    c[i+0] = a[i+0]+b[i+0];
    c[i+1] = a[i+1]+b[i+1];
    c[i+2] = a[i+2]+b[i+2];
    c[i+3] = a[i+3]+b[i+3];
    c[i+4] = a[i+4]+b[i+4];
    c[i+5] = a[i+5]+b[i+5];
    c[i+6] = a[i+6]+b[i+6];
    c[i+7] = a[i+7]+b[i+7];
}
```

- count = 16
- #pragma omp SIMD
- **AVX-512** (flagi -mavx512f -mavx512cd)

Kompilator dzieli pętle na bloki o liczbie iteracji dopasowanej do rozmiaru rejestru oraz użytego typu danych:



np. dla AVX-512 oraz typu double liczba iteracji dla pojedynczego bloku wynosi 8

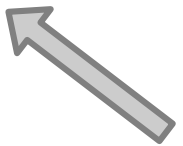
Półautomatyczna wektoryzacja na przykładzie OpenMP

```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i+=8) {
    c[i+0] = a[i+0]+b[i+0];
    c[i+1] = a[i+1]+b[i+1];
    c[i+2] = a[i+2]+b[i+2];
    c[i+3] = a[i+3]+b[i+3];
    c[i+4] = a[i+4]+b[i+4];
    c[i+5] = a[i+5]+b[i+5];
    c[i+6] = a[i+6]+b[i+6];
    c[i+7] = a[i+7]+b[i+7];
}
```

- count = 16
- #pragma omp SIMD
- **AVX-512** (flagi -mavx512f -mavx512cd)



W ramach każdej iteracji, kompilator zamieni osiem instrukcji skalarnych na jedną instrukcję wektorową: vaddpd zmm, zmm, zmm

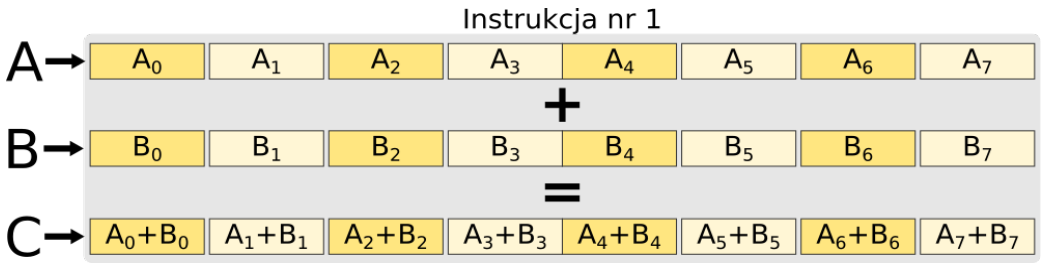
Półautomatyczna wektoryzacja na przykładzie OpenMP

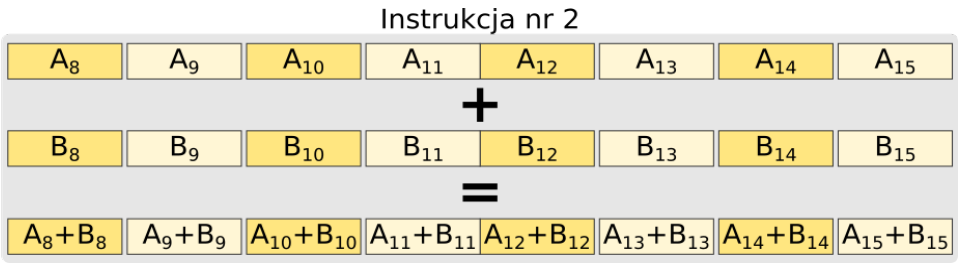
```
double a[n];
double b[n];
double c[n];

(...)

#pragma omp simd
for (i = 0; i < count; i+=8) {
    c[i+0] = a[i+0]+b[i+0];
    c[i+1] = a[i+1]+b[i+1];
    c[i+2] = a[i+2]+b[i+2];
    c[i+3] = a[i+3]+b[i+3];
    c[i+4] = a[i+4]+b[i+4];
    c[i+5] = a[i+5]+b[i+5];
    c[i+6] = a[i+6]+b[i+6];
    c[i+7] = a[i+7]+b[i+7];
}
```

- count = 16
- #pragma omp SIMD
- **AVX-512** (flagi -mavx512f -mavx512cd)





Półautomatyczna wektoryzacja na przykładzie OpenMP

- OpenMP oferuje możliwość wektoryzacji obliczeń
- Dyrektywa OpenMP SIMD zleca kompilatorowi polecenie zwektoryzwoania danej pętli
  - Kompilator ma obowiązek wektoryzji zgodnie z założeniami użytkownika, który jest odpowiedzialny za poprawność definiowanych założeń użytkownika
- Wektoryzacja pętli realizowana jest poprzez dodanie odpowiedniej adnotacji przed pętlą, której kolejno występujące po sobie iteracje zostaną przebudowane

Półautomatyczna wektoryzacja na przykładzie OpenMP

- Składnia:

  #pragma omp simd *[clause[[,] clause] ...]*

  for-loops

- Klauzule:

- **safelen**(length)
- **simdlen**(length)
- **aligned**(list[: alignment])
- **nontemporal**(list)

- **private**(list)
- **lastprivate**([lastprivate-modifier:] list)
- **reduction**([modifier,]identifier: list)
- **collapse**(n)
- **order**(concurrent)

Półautomatyczna wektoryzacja na przykładzie OpenMP

- Składnia:

  #pragma omp simd *[clause[[,] clause] ...]*

  for-loops

- Klauzule:

Podobna funkcjonalność
jak w przypadku ***#pragma omp for***

- **safelen**(length)
- **simdlen**(length)
- **aligned**(list[: alignment])
- **nontemporal**(list)

- **private**(list)
- **lastprivate**([lastprivate-modifier:] list)
- **reduction**([modifier,]identifier: list)
- **collapse**(n)
- **order**(concurrent)

Półautomatyczna wektoryzacja na przykładzie OpenMP

- Składnia:

  #pragma omp simd *[clause[[,] clause] ...]*

  for-loops

- Klauzule:

Podobna funkcjonalność
jak w przypadku ***#pragma omp for***

- **safelen**(length)
- **simdlen**(length)
- **aligned**(list[: alignment])
- **nontemporal**(list)

- **private**(list)
- **lastprivate**([lastprivate-modifier:] list)
- **reduction**([modifier,]identifier: list)
- **collapse**(n)
- **order**(concurrent)

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **safelen(N)**
  - Programista zapewnia kompilator, że nie ma zależności pomiędzy *N* kolejno występujących po sobie iteracjami pętli
  - Domyślna wartość odpowiada maksymalnej liczbie iteracji

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **safelen(N)**
  - Programista zapewnia kompilator, że nie ma zależności pomiędzy *N* kolejno występujących po sobie iteracjami pętli
  - Domyślna wartość odpowiada maksymalnej liczbie iteracji
- W poniższym kodzie, programista gwarantuje, że nie ma zależności dla co najmniej 16 iteracji pętli.
- Dla m<16 program zwróci błędny wynik dla tablicy b

```
void work( float *b, int n, int m ) {
    #pragma omp simd safelen(16)
    for (int i = m; i < n; i++)
        b[i] = b[i-m] - 1.0f;
}
```

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **simdlen(N)**
  - Programista określa preferencje dla maksymalnej długości wektora odpowiadającej liczbie iteracji pętli dla każdego bloku wydzielanego z pętli

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **simdlen(N)**
  - Programista określa preferencje dla maksymalnej długości wektora odpowiadającej liczbie iteracji pętli dla każdego bloku wydzielanego z pętli
- W poniższym kodzie programista definiuje rozmiar wektora odpowiadający łącznemu wykonaniu 16 iteracji pętli
- Zatem, dla typu float oraz wektora składającego się z 16 elementów można zastosować instrukcje AVX-512, AVX, lub SSE

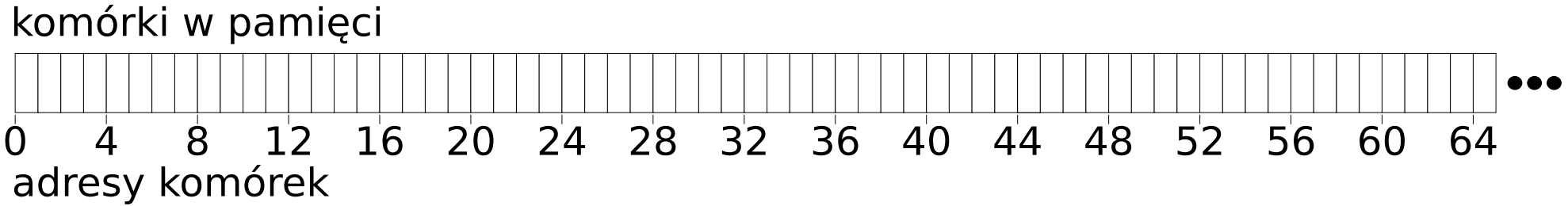
```
void work( float *b, int n) {
    #pragma omp simd simdlen(16)
    for (int i = 0; i < n; i++)
        b[i] = b[i] - 1.0f;
}
```

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy

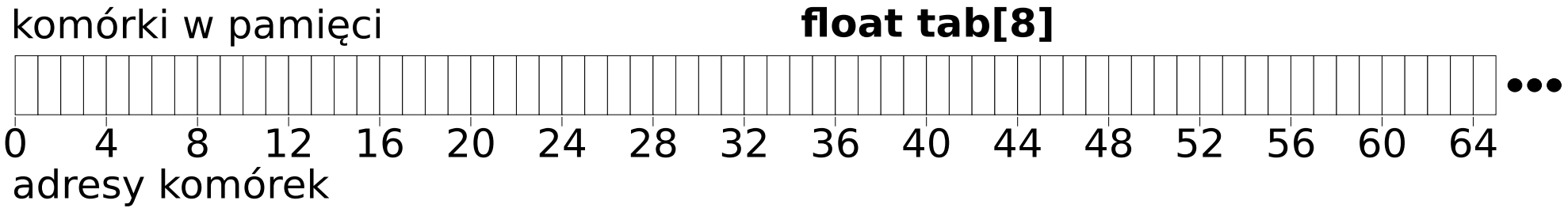
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



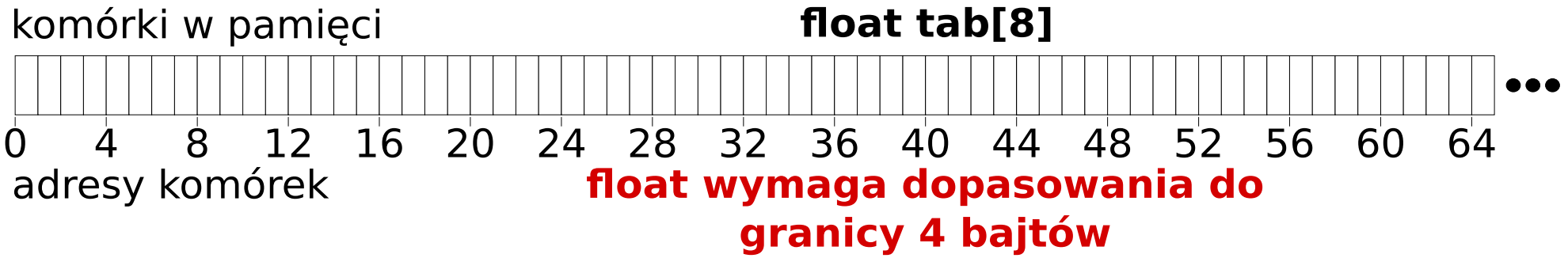
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



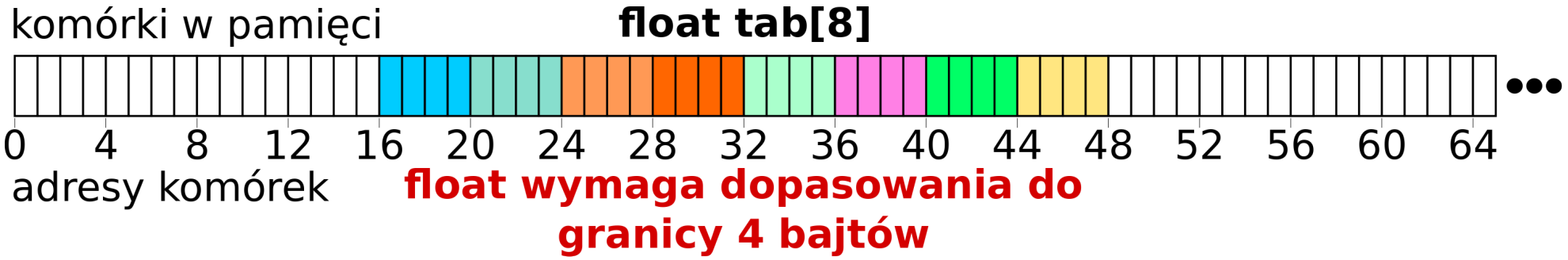
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



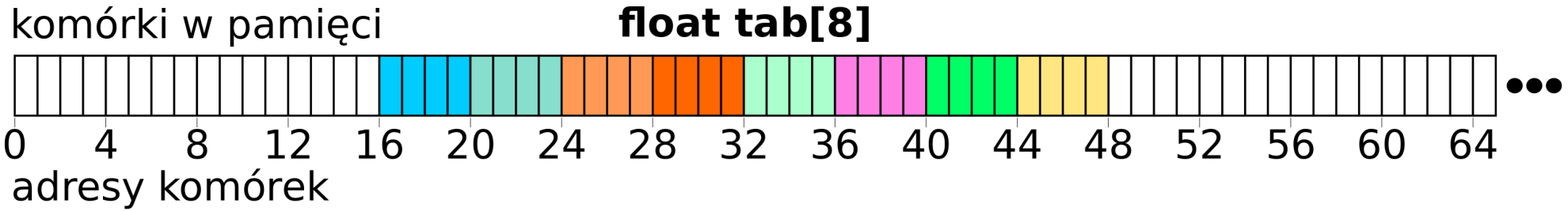
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



Półautomatyczna wektoryzacja na przykładzie OpenMP

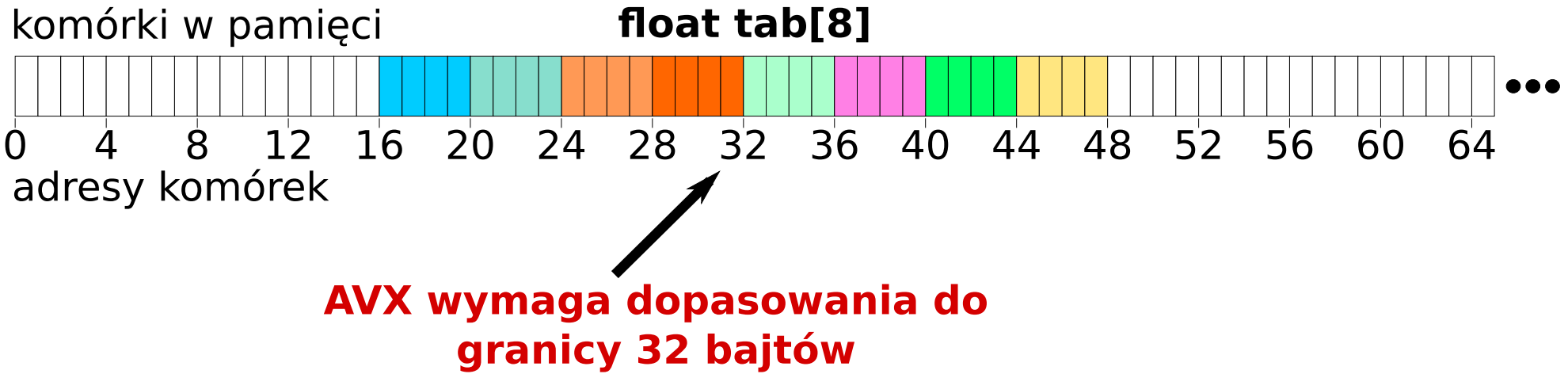
- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



AVX wymaga dopasowania do granicy 32 bajtów

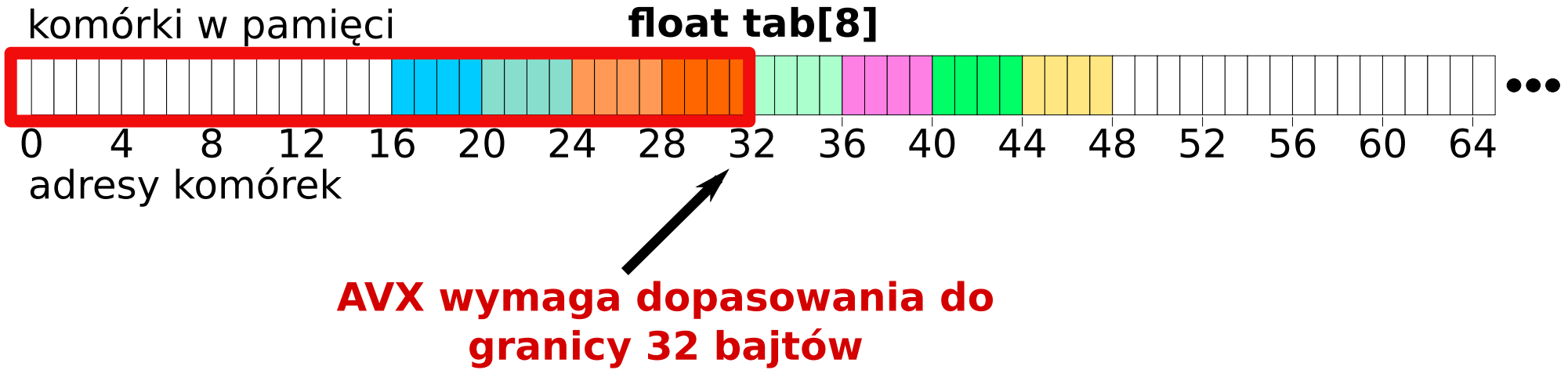
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



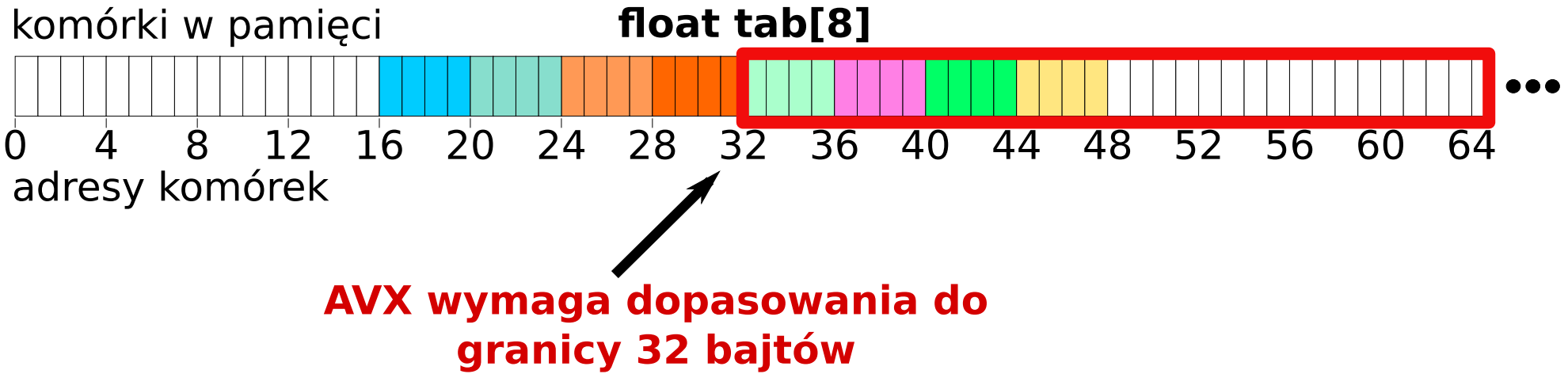
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



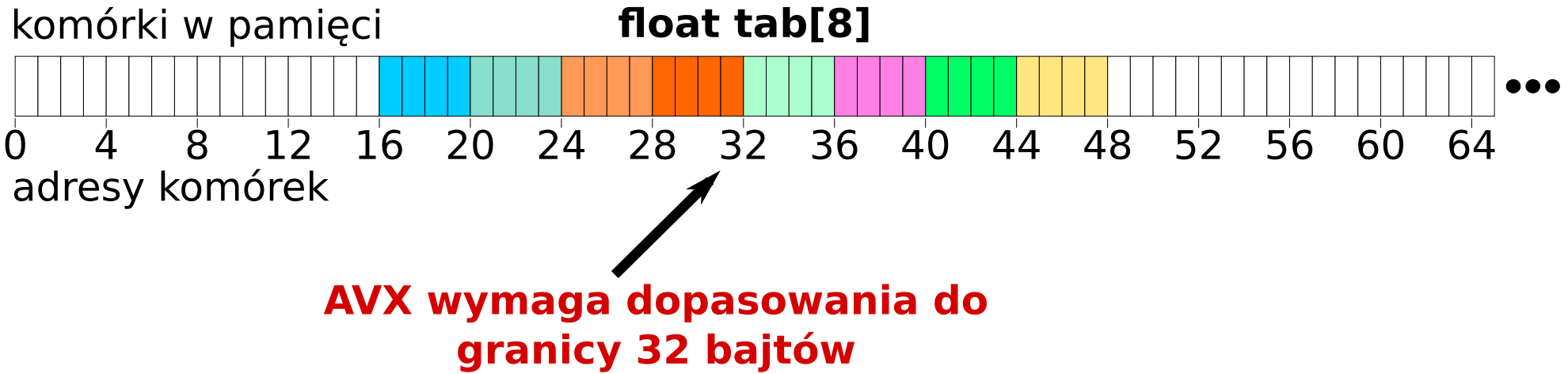
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



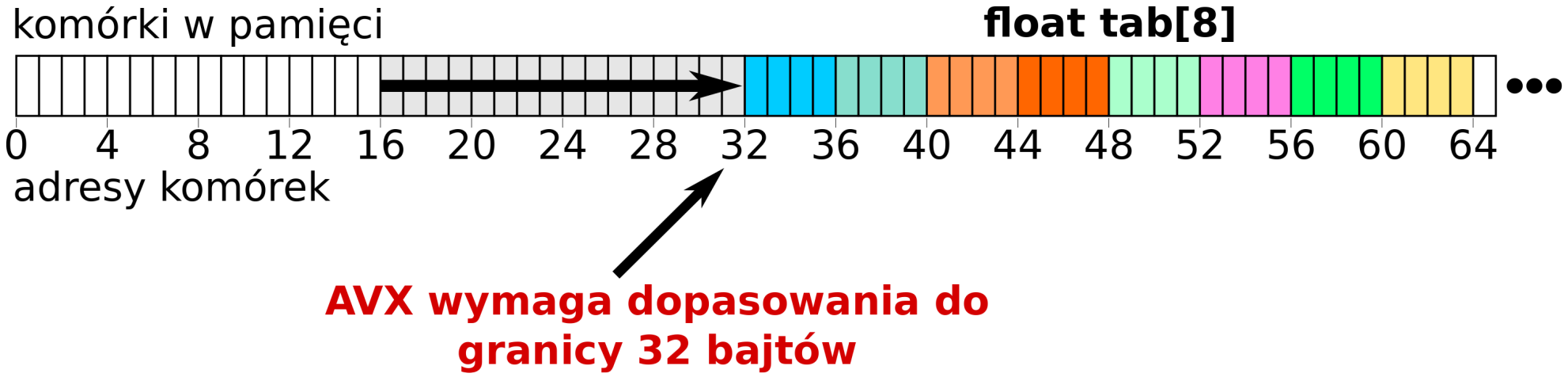
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



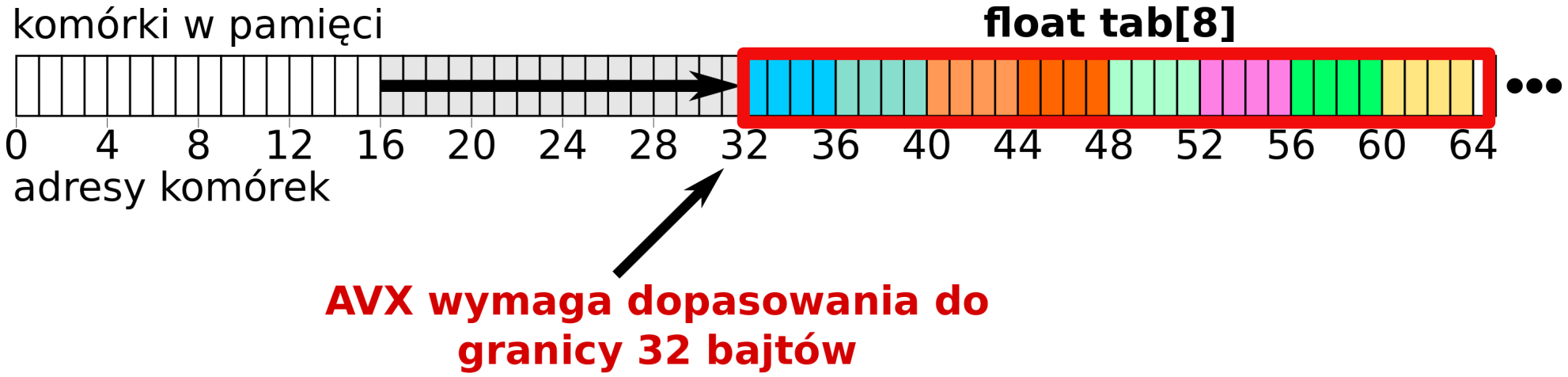
Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy definiowane dla pierwszej iteracji pętli są dopasowane w pamięci głównej do wskazanej granicy



Półautomatyczna wektoryzacja na przykładzie OpenMP

- W przypadku języka C/C++ można wymusić alokację danych zawartych pod adres dopasowanym do wymaganej granicy:
- Dla tablic dynamicznych może skorzystać z kombinacji funkcji:

```
int posix_memalign(void**ptr, size_t alig, size_t size);
void free(void *ptr);
```

- Dla tablic statycznych można skorzystać z następujących atrybutów:

```
__attribute__((aligned( alignment ))
```

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **aligned(list[: alignment])**
  - Programista określa listę zmiennych których adresy są dopasowane w pamięci głównej do wskazanej granicy
  - dla SSE dane powinny być dopasowana do 16 bajtów
  - dla AVX dane powinny być dopasowana do 32 bajtów
  - dla AVX-512 dane powinny być dopasowana do 64 bajtów

Półautomatyczna wektoryzacja na przykładzie OpenMP

- W poniższym kodzie programista zapewnia kompilator, że lista parametrów *a, b i c* odnosi się do adresów tablic dopasowanych do granicy 64 bajtów

```
double a[n] __attribute__((aligned( 64 ));
double b[n] __attribute__((aligned( 64 ));
double c[n] __attribute__((aligned( 64 ));

(...)

#pragma omp simd aligned(a,b,c:64)
for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **nontemporal**(list)
- Klauzula umożliwia bezpośredni zapis wtników obliczeń do pamięci głównej z pominięciem pamięci podręcznej, o ile architektura wspiera tego rodzaju operacji

Półautomatyczna wektoryzacja na przykładzie OpenMP

- **nontemporal**(list)
- Klauzula umożliwia bezpośredni zapis wtników obliczeń do pamięci głównej z pominięciem pamięci podręcznej, o ile architektura wspiera tego rodzaju operacji
- W poniższym kodzie, programista narzuca kompilatorowi zastosowanie operacji bezpośredniego zapisywania wyników obliczeń do pamięci głównej z pominięciem pamięci podręcznej

```
#pragma omp simd aligned(a,b,c:64) nontemporal(c)
for (i = 0; i < count; i++) {
    c[i] = a[i]+b[i];
}
```

Półautomatyczna wektoryzacja na przykładzie OpenMP

- OpenMP umożliwia łączenia dyrektywy zrównoleglania pętli z wykorzystaniem wątków z dyrektywą dla wektoryzacji obliczeń:
- Składnia:

#pragma omp for simd *[clause[[,] clause] ...]*

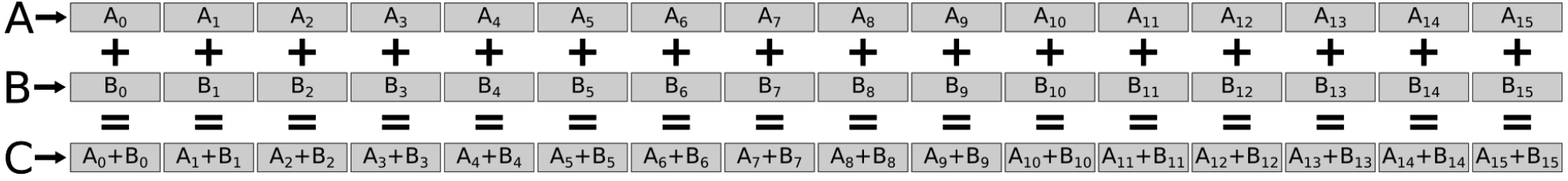
for-loops

Półautomatyczna wektoryzacja na przykładzie OpenMP

```
void vecAdd(double *a, double *a, double *c, int n) {
    #pragma omp for simd aligned(a,b,c:64) nontemporal(c)
    for (i = 0; i < count; i++) {
        c[i] = a[i]+b[i];
    }
}
```

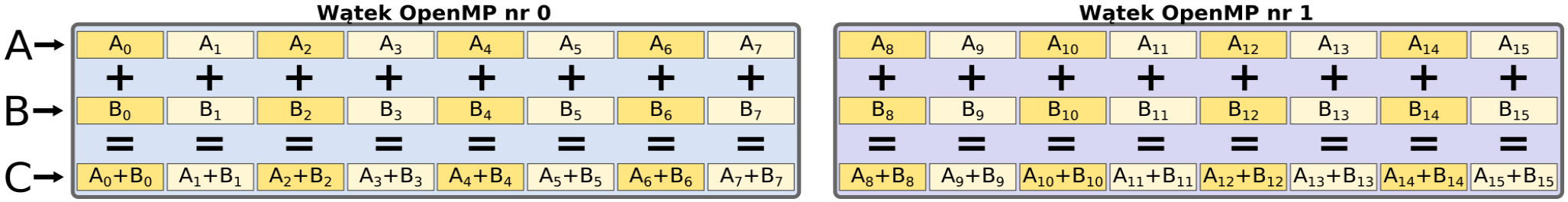
Półautomatyczna wektoryzacja na przykładzie OpenMP

```
void vecAdd(double *a, double *a, double *c, int n) {
    #pragma omp for simd aligned(a,b,c:64) nontemporal(c)
    for (i = 0; i < count; i++) {
        c[i] = a[i]+b[i];
    }
}
```



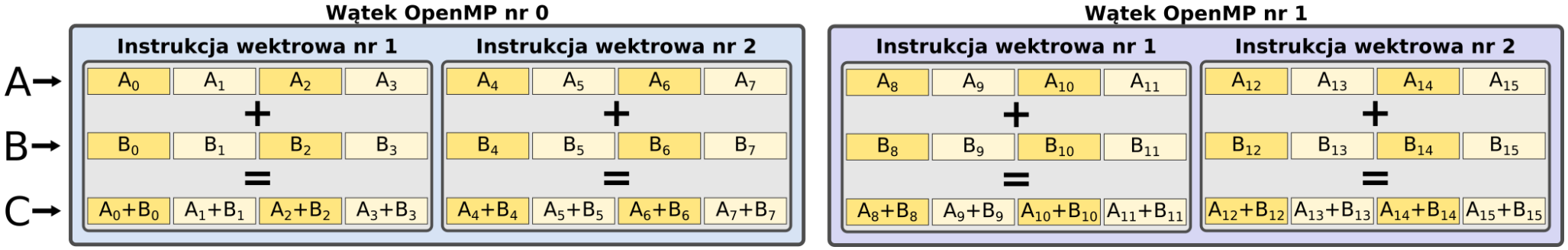
Półautomatyczna wektoryzacja na przykładzie OpenMP

```
void vecAdd(double *a, double *a, double *c, int n) {
    #pragma omp for simd aligned(a,b,c:64) nontemporal(c)
    for (i = 0; i < count; i++) {
        c[i] = a[i]+b[i];
    }
}
```



Półautomatyczna wektoryzacja na przykładzie OpenMP

```
void vecAdd(double *a, double *a, double *c, int n) {
    #pragma omp for simd aligned(a,b,c:64) nontemporal(c)
    for (i = 0; i < count; i++) {
        c[i] = a[i]+b[i];
    }
}
```



Manualna wektoryzacja obliczeń

Manualna wektoryzacja obliczeń

- Instrukcje wektorowe mogą zostać również zaimplementowane bezpośrednio w kodzie źródłowym aplikacji
- Rozwiązanie to zapewnia lepszą kontrolę nad sposobem wykonywania obliczeń, jednakże zazwyczaj okazuje się trudniejsze w użyciu oraz bardziej czasochłonne
- Poprawne zastosowanie wektoryzacji wymaga restrukturyzacji obszarów kodu źródłowego

Manualna wektoryzacja obliczeń

- Ręczna implementacja technik wektoryzacji w programie możliwa jest przy użyciu:
  - instrukcji asemblerowych
  - **instrukcji intrinsic**
  - biblioteki klas wektorowych

Manualna wektoryzacja obliczeń

- Instrukcje intrinsic są funkcjami języków C lub C++ będącymi odpowiednikami niskopoziomowych instrukcji wektorowych
- Funkcje te wykonują operacje na typach danych reprezentujących 512-bitowe, 255-bitowe lub 128-bitowe rejestry
- Pełen zestaw instrukcji intrinsic oraz typów danych można znaleźć tutaj: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Manualna wektoryzacja obliczeń

- Przykładowo, operacja dodawania dwóch wektorów dla danych typu **float** realizowana jest przy pomocy następujących instrukcji wbudowanych:

SSE

__m128 _**mm_add_ps** (__m128 a, __m128 b)
#include <xmmintrin.h>
Instruction: addps xmm, xmm

AVX

__m256 _**mm256_add_ps** (__m256 a, __m256 b)
#include <immintrin.h>
Instruction: vaddps ymm, ymm, ymm

AVX512

__m512 _**mm512_add_ps** (__m512 a, __m512 b)
#include <immintrin.h>
Instruction: vaddps zmm, zmm, zmm

Manualna wektoryzacja obliczeń

- Przykładowo, operacja dodawania dwóch wektorów dla danych typu **double** realizowana jest przy pomocy następujących instrukcji wbudowanych:

SSE2

__m128 _**mm_add_pd** (__m128 a, __m128 b)
#include <emmintrin.h>
Instruction: addpd xmm, xmm

AVX

__m256 _**mm256_add_pd** (__m256 a, __m256 b)
#include <immintrin.h>
Instruction: vaddpd ymm, ymm, ymm

AVX512

__m512 _**mm512_add_pd** (__m512 a, __m512 b)
#include <immintrin.h>
Instruction: vaddpd zmm, zmm, zmm

Manualna wektoryzacja obliczeń

- Przykładowo, operacje na wektorach o rozmiarze 256 bitów dla danych typu **float** realizowana jest przy pomocy następujących instrukcji wbudowanych:

```
__m256 * __restrict__ vecA = (__m256*)&a[0];
__m256 * __restrict__ vecB = (__m256*)&b[0];
__m256 * __restrict__ vecC = (__m256*)&c[0];
__m256 * __restrict__ vecD = (__m256*)&d[0];
for (int i = 0; i < vecN; ++i) {
    __m256 tmp1 = _mm256_mul_ps(vecA[i],vecB[i]);
    __m256 tmp2 = _mm256_add_ps(tmp1,vecC[i]);
    vecD[i] = _mm256_add_ps(tmp1,tmp2);
}
```

- Flagi do kompilacji -O3 -mavx

Manualna wektoryzacja obliczeń

- Przykładowo, operacje na wektorach o rozmiarze 512 bitów dla danych typu **float** realizowana jest przy pomocy następujących instrukcji wbudowanych:

```
__m512 * __restrict__ vecA = (__m512*)&a[0];
__m512 * __restrict__ vecB = (__m512*)&b[0];
__m512 * __restrict__ vecC = (__m512*)&c[0];
__m512 * __restrict__ vecD = (__m512*)&d[0];
for (int i = 0; i < vecN; ++i) {
    __m512 tmp1 = _mm512_mul_ps(vecA[i],vecB[i]);
    __m512 tmp2 = _mm512_add_ps(tmp1,vecC[i]);
    vecD[i] = _mm512_add_ps(tmp1,tmp2);
}
```

- Flagi do kompilacji -O3 -mavx512f -mavx512cd

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**
- Warunkiem wektoryzacja pętli jest wykazanie X kolejnych iteracji pętli, których wykonanie jest niezależne
  - X – oznacza liczbę elementów w wektorze

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i];
}
```

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i-1];
}
```

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i];
}
```

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i];
}
```

```
for(int i=0; i<16; i+=4){
    C[i+0] = A[i+0] * B[i+0] + C[i+0];
    C[i+1] = A[i+1] * B[i+1] + C[i+1];
    C[i+2] = A[i+2] * B[i+2] + C[i+2];
    C[i+3] = A[i+3] * B[i+3] + C[i+3];
}
```

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i];
}
```

```
for(int i=0; i<16; i+=4){
    C[i+0] = A[i+0] * B[i+0] + C[i+0];
    C[i+1] = A[i+1] * B[i+1] + C[i+1];
    C[i+2] = A[i+2] * B[i+2] + C[i+2];
    C[i+3] = A[i+3] * B[i+3] + C[i+3];
}
```

Brak zależności między kolejnymi iteracjami pętli

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i-1];
}
```

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i-1];
}
```

```
for(int i=0; i<16; i+=4){
    C[i+0] = A[i+0] * B[i+0] + C[i+0-1];
    C[i+1] = A[i+1] * B[i+1] + C[i+1-1];
    C[i+2] = A[i+2] * B[i+2] + C[i+2-1];
    C[i+3] = A[i+3] * B[i+3] + C[i+3-1];
}
```

Dlaczego wektoryzacja nie działa?

- **Nie wszystkie pętle można zwektoryzować**

```
for(int i=0; i<16; ++i){
    C[i] = A[i] * B[i] + C[i-1];
}
```

```
for(int i=0; i<16; i+=4){
    C[i+0] = A[i+0] * B[i+0] + C[i+0-1];
    C[i+1] = A[i+1] * B[i+1] + C[i+1-1];
    C[i+2] = A[i+2] * B[i+2] + C[i+2-1];
    C[i+3] = A[i+3] * B[i+3] + C[i+3-1];
}
```

- Wykonanie każdej iteracji zależy od wcześniejszej,
- Przykładowo:

c[4] wymaga c[3]
c[5] wymaga c[4]
c[6] wymaga c[5]

Wektoryzacja obliczeń

Zadanie 1

- Napisz program równoległy z wykorzystaniem jednostek wektorowych umożliwiający wyznaczenie sumy wszystkich elementów dwuwymiarowej tablicy
- Spróbuj zrealizować zadania na trzy sposoby z wykorzystaniem automatycznej, półautomatycznej oraz manualnej wektoryzacji obliczeń
- Dodatkowo zastosuj zrównoleglanie programu z użyciem wątków OpenMP

Projekt w ramach programu Regionalna Inicjatywa Doskonałości, decyzja nr 020/RID/2018/19

Dziękuję za uwagę!