**\_PODPOWIEDZI**

Oczekuje się, że wersja SIMD (Single Instruction, Multiple Data) będzie znacznie szybsza niż wersja SISD (Single Instruction, Single Data) z następujących powodów:

* + Równoległość na poziomie danych: Instrukcje SIMD (takie jak mulps użyte w kodzie) są zaprojektowane do wykonywania tej samej operacji (w tym przypadku mnożenia) na wielu elementach danych jednocześnie. W dostarczonym kodzie SSE, jedna instrukcja mulps mnoży cztery pary liczb zmiennoprzecinkowych równocześnie.
  + Mniejsza liczba instrukcji: Aby przetworzyć tę samą ilość danych, wersja SIMD wykonuje znacznie mniej instrukcji procesora. Wersja SISD (używająca x87 FPU) musi załadować, pomnożyć i zapisać każdy element danych indywidualnie, co wymaga osobnego zestawu instrukcji dla każdego elementu.
  + Efektywniejsze wykorzystanie jednostek wykonawczych procesora: Nowoczesne procesory mają dedykowane, szerokie jednostki wykonawcze SIMD, które mogą przetwarzać 128 bitów (lub więcej, np. 256 bitów dla AVX, 512 bitów dla AVX-512) danych w jednym cyklu zegarowym.

Wersja SISD z x87 FPU, mimo że jest to wyspecjalizowana jednostka zmiennoprzecinkowa, operuje na pojedynczych wartościach i często wiąże się z większym narzutem na zarządzanie stosem FPU w porównaniu do operacji na rejestrach XMM używanych przez SSE.

**Jakie są wyzwania lub kwestie do rozważenia dla każdego z nich (np. zarządzanie stosem FPU, zarządzanie rejestrami, wyrównanie, szczegółowość kodu, przenośność)?**

* x87 FPU (dla SISD):  
  + Zarządzanie stosem: Największym wyzwaniem jest zarządzanie stosem FPU (st(0), st(1) itd.). Programista musi śledzić, które wartości znajdują się na których pozycjach stosu, i używać odpowiednich instrukcji do ładowania, operowania i zdejmowania wartości ze stosu (fld, fmul, fstp, fmulp, fstps). Błędy w zarządzaniu stosem mogą prowadzić do trudnych do zdiagnozowania problemów.
  + Mniejsza liczba rejestrów (efektywnie): Chociaż jest 8 rejestrów FPU, ich stosowa natura ogranicza elastyczność w porównaniu do płaskiej przestrzeni rejestrów XMM.
  + Precyzja: Rejestry x87 FPU wewnętrznie używają 80-bitowej precyzji rozszerzonej, nawet jeśli operacje są na liczbach pojedynczej (32-bit) lub podwójnej (64-bit) precyzji. Może to czasami prowadzić do nieco innych wyników niż przy użyciu SSE, które ściśle operuje na precyzji zadeklarowanej (np. 32-bit dla float).
  + Szczegółowość kodu: Kod może być dość szczegółowy z powodu operacji na stosie.
  + Wyrównanie: Zazwyczaj mniej problematyczne dla pojedynczych operacji skalarnych.
  + Przenośność: x87 FPU jest starszą technologią, ale nadal szeroko wspieraną. Jednak nowoczesne kompilatory i systemy operacyjne mogą preferować użycie SSE dla operacji skalarnych ze względu na wydajność i spójność z operacjami wektorowymi.
* SSE (dla SIMD):  
  + Zarządzanie rejestrami: Programista ma do dyspozycji zestaw płaskich rejestrów XMM (np. xmm0 - xmm15 w trybie 64-bit). Jest to bardziej bezpośrednie niż zarządzanie stosem.
  + Wyrównanie danych: To kluczowe wyzwanie. Wiele instrukcji SSE (np. movaps, mulps) wymaga, aby adresy pamięci, z których dane są ładowane lub do których są zapisywane, były wyrównane do 16 bajtów. Niewyrównany dostęp może prowadzić do błędów wykonania (segmentation fault) lub znacznego spadku wydajności (jeśli używane są wolniejsze instrukcje niewyrównane, np. movups). W kodzie użyto aligned\_alloc i \_\_attribute\_\_((aligned(16))) do zapewnienia wyrównania.
  + Przygotowanie danych: Dane często muszą być specjalnie przygotowane (np. rozgłoszenie skalara do wszystkich elementów wektora SIMD za pomocą shufps).
  + Obsługa resztek: Jeśli liczba elementów danych nie jest idealną wielokrotnością liczby elementów, które SIMD może przetworzyć naraz (np. 4 dla SSE), trzeba zaimplementować dodatkową logikę (np. pętlę skalarną) do obsługi pozostałych elementów. W tym laboratorium rozmiar tablicy jest wielokrotnością 4, aby uprościć ten aspekt.
  + Szczegółowość kodu: Może być większa niż w przypadku prostych operacji skalarnych w C, ale często bardziej zwięzła niż x87 dla tej samej ilości przetworzonych danych ze względu na przetwarzanie wielu elementów naraz.
  + Przenośność: SSE jest szeroko dostępne na nowoczesnych procesorach x86. Nowsze rozszerzenia (AVX, AVX2, AVX-512) oferują jeszcze większą moc, ale wymagają nowszych procesorów.

**Zmiana NUM\_ELEMENTS:**

* **Zwiększenie NUM\_ELEMENTS:** Generalnie, im większy jest zbiór danych (NUM\_ELEMENTS), tym bardziej widoczne stają się korzyści z SIMD. Narzut związany z inicjalizacją pętli i przygotowaniem danych SIMD (np. rozgłaszanie skalara) jest amortyzowany przez większą liczbę operacji. Dla bardzo małych NUM\_ELEMENTS, narzut ten mógłby teoretycznie sprawić, że SIMD byłoby wolniejsze lub tylko nieznacznie szybsze. Dla dużych NUM\_ELEMENTS (jak w laboratorium), przyspieszenie SIMD powinno być znaczące i zbliżone do teoretycznego maksimum (np. 4x dla SSE przetwarzającego 4 floaty).
* **Zmniejszenie NUM\_ELEMENTS:** Przyspieszenie może spaść, ponieważ stały narzut SIMD staje się bardziej znaczący w stosunku do całkowitego czasu przetwarzania.

**Poziomy optymalizacji kompilatora (-O0, -O1, -O2, -O3):**

* Kompilator odgrywa kluczową rolę w generowaniu wydajnego kodu maszynowego. Różne flagi optymalizacji instruują kompilator, aby zastosował różne techniki w celu poprawy szybkości i/lub zmniejszenia rozmiaru kodu
* **-O0 (Bez optymalizacji):**
  + Kompilator stara się jak najwierniej odwzorować kod źródłowy na kod maszynowy, co ułatwia debugowanie.
  + Kod jest zazwyczaj znacznie wolniejszy, ponieważ nie są stosowane żadne zaawansowane techniki optymalizacyjne (np. rozwijanie pętli, eliminacja wspólnych podwyrażeń, alokacja rejestrów).
  + W przypadku kodu z asemblerem, wpływ -O0 na sam blok \_\_asm\_\_ jest ograniczony (zwłaszcza z volatile), ale otaczający kod C (np. pętle, zarządzanie zmiennymi) będzie nieoptymalizowany.
  + **Wpływ na przyspieszenie:** Trudno przewidzieć jednoznacznie. Obie wersje (SISD i SIMD) będą wolniejsze. Jeśli nieoptymalizowany kod pętli C bardziej szkodzi wersji SISD (która ma więcej iteracji pętli dla tej samej ilości danych), przyspieszenie SIMD może wydawać się większe. Jednakże, jeśli narzuty związane z wywołaniem funkcji lub zarządzaniem pamięcią są duże w nieoptymalizowanym kodzie, mogą one wpłynąć na obie wersje.
* **-O1 (Podstawowe optymalizacje):**
  + Włącza podstawowe optymalizacje, które nie wydłużają znacząco czasu kompilacji. Obejmuje to np. eliminację martwego kodu, prostą alokację rejestrów.
  + Kod będzie szybszy niż przy -O0.
  + **Wpływ na przyspieszenie:** Prawdopodobnie przyspieszenie SIMD będzie bardziej reprezentatywne dla rzeczywistych korzyści niż przy -O0.
* **-O2 (Większość optymalizacji):**
  + Włącza prawie wszystkie optymalizacje, które nie obejmują kompromisu między szybkością a rozmiarem kodu (space-speed tradeoff). Jest to często zalecany poziom optymalizacji dla wydań produkcyjnych. Obejmuje bardziej zaawansowane techniki, takie jak instrukcje szeregujące (instruction scheduling), lepsze rozwijanie pętli.
  + Kod jest zazwyczaj znacznie szybszy niż przy -O1.
  + **Wpływ na przyspieszenie:** Powinien dać dobre oszacowanie praktycznego przyspieszenia SIMD. Kompilator może być w stanie lepiej zoptymalizować kod C otaczający bloki asemblera.
* **-O3 (Najwyższe optymalizacje):**
  + Włącza wszystkie optymalizacje z -O2 oraz dodatkowe, które mogą być bardziej agresywne i czasami mogą zwiększyć rozmiar kodu lub nawet (w rzadkich przypadkach, dla specyficznych kodów) nieznacznie pogorszyć wydajność niektórych części, jednocześnie poprawiając inne. Może obejmować bardziej agresywne rozwijanie pętli, wektoryzację automatyczną (choć w naszym przypadku mamy jawną wektoryzację SIMD w asemblerze), czy inlinowanie funkcji.
  + Często daje najszybszy kod, ale czas kompilacji jest dłuższy.
  + **Wpływ na przyspieszenie:** Może nieznacznie zwiększyć przyspieszenie SIMD, jeśli kompilator znajdzie dodatkowe sposoby na optymalizację kodu C, zwłaszcza jeśli uda mu się lepiej zoptymalizować pętlę SISD lub narzuty związane z wywołaniami funkcji. Jednakże, ponieważ kluczowe fragmenty są już w asemblerze liniowym, wpływ na te konkretne bloki będzie minimalny. Główny wpływ będzie na kod C generowany przez kompilator *wokół* bloków asemblera.

**Podsumowując wpływ optymalizacji na przyspieszenie:** Generalnie, wyższe poziomy optymalizacji (-O2, -O3) powinny skutkować szybszym wykonaniem zarówno kodu SISD, jak i SIMD (w częściach zarządzanych przez kompilator). Przyspieszenie SIMD nad SISD jest miarą względną. Jeśli optymalizacje kompilatora przynoszą proporcjonalnie większe korzyści dla kodu SISD (np. przez lepsze zarządzanie pętlą w C), to zmierzone przyspieszenie SIMD może nieznacznie spaść przy wyższych poziomach optymalizacji. Jeśli jednak optymalizacje bardziej pomagają zminimalizować narzuty w kodzie SIMD, przyspieszenie może wzrosnąć. Najczęściej oczekuje się, że przy -O2 lub -O3 uzyskane przyspieszenie będzie najbardziej realistycznym odzwierciedleniem korzyści płynących z ręcznie zaimplementowanego SIMD.