Tematem niniejszego tekstu jest, jak głosi tytuł, kilka technik optymalizacji programów napisanych w języku C, celem zwiększenia ich wydajności. Ale zanim przejdziemy do omawiania tej problematyki, może warto na wstępie przypomnieć jedną z podstawowych zasad dotyczących tego zagadnienia, tę mianowicie, która głosi, że do optymalizacji programu przystępujemy zawsze dopiero wtedy, kiedy jest on przynajmniej w swojej głównej (obliczeniowej) części już gotowy, kompletny i przetestowany pod kątem prawidłowego działania. To pozwala zidentyfikować wszelkie anomalie powstałe później w wyniku optymalizacji, na skutek czy to błędów popełnionych przy tym przez programistów, czy to zbyt agresywnych w tej kwestii poczynań kompilatora.

Programiści muszą się też oczywiście dobrze zapoznać z odnośnymi funkcjami narzędzi deweloperskich, co pozwoli im potem zaoszczędzić sporo czasu i wysiłku, gdyż postęp w dziedzinie kompilatorów powoduje, że są one coraz lepsze, a tym samym potrafią coraz lepiej odciążyć programistę w procesie tworzenia dobrego kodu.

Nie jest jednak tak, jak głosi niekiedy spotykana opinia, że w dzisiejszych czasach programista nie jest w stanie zrobić z kodem źródłowym już niczego, co spowodowałoby polepszenie jakości kodu wynikowego, że kompilatory optymalizują kod w sposób doskonały i że człowiek, nawet pisząc daną procedurę bezpośrednio w asemblerze, nie jest w stanie zrobić tego lepiej, gdyż opiera się na tym samym zasobie wiedzy o danej architekturze, z którego korzysta także kompilator.

To ostatnie jest, oczywiście, prawdą, ale przedstawione wnioski, które się z niej wyciąga, poprawne nie są, i to aż z dwóch powodów. Po pierwsze, mimo że składnia **języka C** służy m.in. do tego, żeby przekazać kompilatorowi intencje programisty, nie przekazuje ich ona w stu procentach: kompilator musi w procesie kompilacji robić pewne założenia i często muszą się one cechować konserwatyzmem polegającym na literalnym traktowaniu tekstu źródłowego programu. Natomiast sama składnia **języka C** sugeruje programistom używanie konstrukcji naturalnych dla człowieka, lecz niekoniecznie optymalnych w danej sytuacji dla danego mikroprocesora. Banalny przykład - z dwóch pętli, jednej takiej:

```
int x;
for (x=0; x<32767; x++)
evaluate(x);
drugiej takiej:
unsigned short int x;
for (x=32766; x>=0; x--)
evaluate(x);
```

człowiek intuicyjnie wybierze tę pierwszą jako prostszą, natomiast po bliższym zbadaniu cech danej architektury może się okazać, że to ta druga jest szybsza - co jednak widać dopiero na poziomie kodu w asemblerze. Ale licznik pętli przyjmuje w obu przypadkach inne wartości, co pociąga za sobą również inne wartości parametru przekazywane do wywoływanej w tej pętli, przykładowej funkcji.

Jeśli na dodatek funkcja evaluate() została zadeklarowana jako extern, kompilator nie jest w stanie zoptymalizować pętli w ten sposób - może to zrobić tylko człowiek. Tylko człowiek może też odpowiednio zmodyfikować funkcję evaluate() i ocenić, czy cała operacja się w ogóle opłaca, tzn. czy zyski z tej optymalizacji nie spowodują większych strat wydajnościowych w innym miejscu.

I tu dochodzimy do następnej przewagi programisty nad kompilatorem: kompilator analizuje program cząstkowo jako serię instrukcji i powodowanych przez nie tak zwanych "skutków ubocznych". Natomiast człowiek jest w stanie ogarnąć program całościowo pod kątem zadań, jakie ma wypełniać oraz warunków, w jakich ma działać.

Optymalizacja, w którą ma być zaangażowany programista, rzadko kiedy sprowadza się do trywialnych czynności typu "zamień dzielenie na przesunięcie bitowe w prawo, bo tak jest szybciej" - to z całą pewnością może zrobić kompilator i najczęściej takie właśnie rzeczy robi. Nie wspominając już o zmianie algorytmu na inny (czego kompilator naturalnie nie jest w stanie zrobić w żaden sposób), sytuacja, w której musi wkroczyć człowiek, wiąże się na ogół z podjęciem jakiejś trudnej decyzji, dajmy na to: przyspieszenia programu w zamian za jakieś koszty poniesione w innym jego miejscu.

Czy te koszty można i warto ponosić, właściwie ocenić jest w stanie tylko twórca programu, bo np. tylko on wie, czy dana pętla będzie wykonywana bardzo często, i w związku z tym warto ją rozwinąć, czy też raczej nie. Krótko mówiąc, kompilator jest narzędziem wyręczającym programistę w wielu mniej lub bardziej mozolnych czynnościach, ale zdarza się, że skuteczna interwencja w program wymaga przeprowadzenia jakichś badań, testów, prób i w ogóle całościowego oglądu sytuacji, a takową kompilator, z natury rzeczy, nie dysponuje.

Optymalizacje w kompilatorze

Próbą częściowego przynajmniej zaradzenia owemu brakowi całościowego oglądu programu przez kompilator jest stosowana czasem technika tak zwanej kompilacji globalnej. Tradycyjny proces kompilacji - który można, dla odróżnienia, nazwać modularną - polega na tym, że każdy z modułów źródłowych programu jest oddzielnie przetwarzany przez preprocesor, kompilowany, optymalizowany i asemblowany do postaci modułu binarnego, a wszystkie uzyskane w ten sposób moduły wynikowe ulegają na końcu konsolidacji.

Dostrzegalną słabością całego tego procesu jest to, że kompilator potrafi dokonać optymalizacji wewnątrz poszczególnych modułów, ale nie ma wpływu na odwołania międzymodułowe do funkcji i zmiennych typu extern - tu musi się ściśle trzymać podanych w kodzie źródłowym definicji. Na przykład, jeśli funkcja jest zadeklarowana w obrębie danego modułu i zostaje wywołana wewnątrz tego modułu, kompilator może ją zupełnie bezpiecznie "wkleić" w miejsce wywołania, likwidując przede wszystkim narzut związany normalnie z wywołaniem funkcji (o czym dalej jeszcze będzie mowa).

Natomiast jeśli ta sama funkcja została zadeklarowana w module innym niż ten, który jej używa, kompilator nie jest w stanie z tym zrobić niczego mądrzejszego niż wygenerowanie zwykłego wywołania funkcji, z wszystkimi tego konsekwencjami.

W każdym razie, kompilacja globalna wprowadza tu drobną, ale doniosłą w skutkach modyfikację: moduły źródłowe są w pierwszej kolejności przetwarzane przez preprocesor, który, oprócz wszystkiego innego, łączy je w jeden, wielki plik przeznaczony dla kompilatora. A dopiero potem ten plik się kompiluje, optymalizuje, asembluje i poddaje konsolidacji w zwykły sposób.

Różnica polega na tym, że przy takim trybie postępowania w programie nie mają już prawa istnieć odwołania międzymodułowe do zmiennych i funkcji typu extern - bo preprocesor na samym wstępie skleja wszystko w jeden moduł. Co za tym idzie, nawet nasza przykładowa funkcja, wywoływana w jednym module, ale zdefiniowana w innym, zostanie teraz poprawnie zidentyfikowana jako taka i - o ile pozwalają na to inne warunki - ładnie "wklejona" w miejsce wywołania.

Niektóre źródła twierdzą, że ta zaleta tej metody stanowi jednocześnie jej wadę: ponieważ kompilator "widzi" wszystkie funkcje i zmienne, może w procesie optymalizacji dokonywać dowolnych ich przetasowań w obrębie całego programu, co spowoduje, że treść tekstu źródłowego straci proste przełożenie na kod wynikowy, a co za tym idzie, program, gdyby miała zajść taka konieczność, będzie trudniejszy do prześledzenia pod debuggerem.

Ale jest to wada pozorna, gdyż dokładnie ta sama trudność powstaje przy tradycyjnej kompilacji (modularnej), toteż, jeśli zachodzi potrzeba użycia debuggera, normalną praktyką jest przekompilowanie przedtem całego programu z wyłączeniem jakichkolwiek automatycznych optymalizacji. Niewątpliwe jest natomiast, że jeden wielki plik wynikowy stosunkowo znacząco wydłuży czas kompilacji; ale czy owo "stosunkowo" będzie oznaczało również, że cały proces stanie się uciążliwy, to już zależy od konkretnego programu i decyzję w tej sprawie należy pozostawić programistom.

Podstawowe ustawienia

Abstrahując od samych opcji optymalizatora, trzeba zwrócić też nieco uwagi na ustawienia kompilatora dotyczące danej architektury. Być może kompilator jest optymalnie skonfigurowany przez producenta, a być może nie: jeśli producent oferuje więcej modeli z jednej rodziny mikroprocesorów, może być tak, że oprogramowanie deweloperskie skonfigurowano pod jakiś wspólny mianownik (tak jak w przypadku rodziny x86 tym wspólnym mianownikiem jest Pentium).

Warto to sprawdzić i w razie potrzeby poprawić konfigurację, żeby program mógł w pełni wykorzystać wszystkie cechy konkretnego CPU. Rzeczy, na które trzeba tu zwrócić uwagę, to przede wszystkim wybrany model mikroprocesora, czy architektura jest little, czy big endian (a może da się to skonfigurować), jaki wybrano model pamięci i czy jest on optymalny dla danej architektury oraz naszych potrzeb.

Do opcji samego optymalizatora zaglądamy na końcu: jak to już objaśniono na wstępie, początkowo, to jest dopóki program nie osiągnie stanu zadowalającej używalności, wszystkie optymalizacje należy wyłączyć. Kiedy są one wyłączone, kompilator w zasadzie dosłownie przekłada kod źródłowy w **języku C** na kod w asemblerze, z zachowaniem wszystkich funkcji, zmiennych, dostępów do nich oraz kolejności wszystkich obliczeń.

Taki kod jest nieefektywny i nie powinno się go używać w produkcyjnych wersjach oprogramowania, ale podczas pisania programu znacznie ułatwia to jego śledzenie przy użyciu debuggera i usuwanie błędów oraz przeróżne przeprowadzane przy tym testy. Na przykład, programista może chcieć zatrzymać program i zmodyfikować wartość zmiennej - tymczasem w kodzie wygenerowanym z włączonymi optymalizacjami tej zmiennej może już wcale nie być, bo została uznana przez kompilator za zbędną i usunięta.

Poziomy optymalizacji są oczywiście różne w kompilatorach różnych producentów, ale da się wyróżnić pięć ich podstawowych wartości:

- O0: optymalizacja wyłączona,
- O1: wysokopoziomowa optymalizacja obliczeń, niezależna od architektury,
- O2: jak -O1, plus optymalizacje zależne od architektury,
- Os: jak -O2, ale z naciskiem na minimalizację rozmiaru kodu wynikowego (s = size),
- O3: jak -O2, ale z naciskiem na maksymalizację szybkości obliczeń kodu wynikowego; włączone zostaje kilka opcji, które mają zwiększać wydajność obliczeniową generując przy tym pewne koszty, np. kompilator będzie próbował rozwijać pętle.

To są ustawienia globalne, ale większość kompilatorów pozwala również na ich lokalną regulację w obrębie poszczególnych modułów, co pozwala na wybór różnego poziomu optymalizacji dla różnych funkcji. A nawet jeśli nasz kompilator jest pozbawiony takich luksusów, zawsze można wybrać różne poziomy optymalizacji dla różnych modułów: często stosowaną praktyką jest ustawienie -O3 dla modułów zawierających części krytyczne pod względem wydajnościowym, a -Os dla np. kodu obsługującego interfejs użytkownika.

Kiedy nie jesteśmy do końca pewni, czy dany moduł powinno się optymalizować na szybkość, czy raczej na rozmiar, zawsze można skorzystać z profilera: jest to narzędzie pozwalające przeanalizować program pod kątem miejsc, w których CPU spędza najwięcej czasu. W każdym razie warto podkreślić, że - co czasem czynią początkujący - na pewno nie należy włączać -O3 dla całego programu, bo może to znacznie powiększyć jego rozmiary, rozwijając pętle również tam, gdzie jest to zupełnie niepotrzebne i nie daje mierzalnych (albo wręcz żadnych) zysków na czasie wykonywania programu.

Zapoznanie się z architekturą

Jednym z kroków, których programista nie może pominąć, jest zapoznanie się z architekturą danego urządzenia w celu pozyskania informacji na temat jej możliwości i ograniczeń. Bez tego nie można w ogóle zresztą myśleć nawet o poprawnym podregulowaniu kompilatora w kierunku generowania przezeń kodu optymalnego dla danego procesora (o czym była mowa w poprzednim akapicie). Przed przystąpieniem do pracy programista powinien zadać sobie kilkanaście istotnych pytań, na przykład:

Czy procesor dysponuje rozkazami dzielenia i mnożenia? Czy istnieją rozkazy mnożenia i akumulacji (MAC)? Czy istnieje wsparcie dla arytmetyki nasyceniowej? Jakie są podstawowe rozmiary słowa: 8, 16, 32 bity? Czy procesor wspiera (lub może natywnie wykonuje) operacje zmiennoprzecinkowe?

A jeśli nie, to czy przynajmniej istnieje wsparcie dla arytmetyki stałopozycyjnej? Czy istnieje wsparcie dla operacji SIMD? Czy kompilator sam reorganizuje kod w celu jego paralelizacji, czy może trzeba się w tym celu odwoływać do funkcji pierwotnych? Ile jest rejestrów i jakiej są wielkości? Czy są to rejestry ogólnego przeznaczenia, czy mają wyspecjalizowane funkcje?

Czy istnieje sprzętowe wsparcie dla pętli? Czy istnieją w danej architekturze predykaty i jakie? Ile jest rodzajów dostępnej pamięci i jakie są między nimi różnice? Ile jest magistral? Ile zapisów i odczytów można przeprowadzić jednocześnie? Czy istnieje możliwość adresowania z odwróceniem bitów (ang. bit-reversal)? itp.

Podstawowe techniki optymalizacji

Wszystkie te czynności wstępne służą jednemu celowi: upewnieniu się, że kod generowany przez kompilator będzie wykorzystywał cechy dostępnej maszyny w stopniu największym z możliwych. Powstaje przy tym kwestia, jak ułożyć kod źródłowy tak, żeby zasugerować kompilatorowi pewne rzeczy, których się normalnie w **języku C** nie sygnalizuje.

Jedną z kluczowych decyzji, które trzeba podjąć, zanim zacznie się pisać program, jest wybór właściwych typów danych. Kompilator ma obowiązek zaakceptowania wszystkich typów przewidzianych w specyfikacji **języka**, ale mogą istnieć powody - głównie wydajnościowe - dla których programiście opłaci się niektóre z nich preferować.

Na przykład, może być tak, że mikroprocesor dysponuje rozkazami mnożenia, ale tylko wartości 16-bitowych. Zatem mnożenie z użyciem 32-bitowych zmiennych całkowitych spowoduje wygenerowanie całej sekwencji rozkazów. Jeśli w danym miejscu nie operujemy na wartościach, które są rzeczywiście 32-bitowe, dużo korzystniej będzie przekształcić wyrażenie tak, żeby kompilator mógł załatwić to mnożenie jednym rozkazem. Podobnie sprawa wygląda z użyciem zmiennych 64-bitowych na procesorach czysto 32-bitowych itp.

Użycie funkcji pierwotnych

Funkcje pierwotne - po ang. built-in functions lub intrinsic functions - to metoda na zapis w **języku C** konstrukcji, które są niemożliwe lub niewygodne do zapisania w normalny sposób. Często są to konstrukcje charakterystyczne dla implementacji kompilatora na konkretnej architekturze - a tym samym, co warto wziąć pod uwagę, mogą być nieprzenośne.

Funkcje takie, w połączeniu z odpowiednio zdefiniowanymi strukturami danych, mogą udostępniać specyficzne dla danej architektury typy danych oraz instrukcje, których nie da się wygenerować przy użyciu ANSI-C. Funkcji pierwotnych używamy tak, jak zwykłych wywołań funkcji, ale kompilator zastępuje je wprost rozkazami mikroprocesora, nie ma zatem narzutu wywołania.

Słowem, w pewnym sensie, funkcje pierwotne są raczej zbliżone do makrodefinicji bardziej niż do prawdziwych funkcji (aczkolwiek trzeba zauważyć, że od momentu wprowadzenia do **języka C** modyfikatora inline różnica pomiędzy funkcją a makrodefinicją uległa niejakiemu rozmyciu).

Za pośrednictwem funkcji pierwotnych możemy zyskać dostęp do kontroli przerwań, arytmetyki nasyceniowej, arytmetyki stałopozycyjnej itd. Przykładowo, filtr o skończonej odpowiedzi impulsowej (FIR) można zaimplementować przy użyciu funkcji pierwotnej w następujący sposób:

```
short fir(short *x, short *y)
{
int i;
long acc;
acc = 0;
for (i=0; i<16; i++)
acc = L_mac(acc, x[i], y[i]);
return (short)acc>>16;
}
```

L_mac() to funkcja pierwotna mnożenia i akumulacji: zastępuje ona oddzielne operacje mnożenia i dodawania, a ponadto działa na zasadzie arytmetyki nasyceniowej, dając pewność, że podczas obliczeń nie nastąpi przepełnienie, a wynik nie znajdzie się poza dopuszczalnym zakresem.

Narzut wywołania funkcji

Metoda wywoływania funkcji jest zależna od architektury. Argumenty funkcji przeważnie przekazywane są przez stos, ale niekiedy w rejestrach, a czasami mamy do czynienia z jakąś kombinacją jednego i drugiego. Domyślna konwencja wywołania funkcji może też być zwykle zmieniona, jeśli programista sobie tego zażyczy, co więcej, ta zmiana może dotyczyć tylko wybranych funkcji, które z jakichś powodów nie powinny być wywoływane standardowo. Może to dotyczyć np. funkcji z bardzo dużą liczbą parametrów, funkcji krytycznych czasowo itp. - w takim układzie może się okazać korzystne przekazywanie przynajmniej części parametrów w rejestrach zamiast na stosie.

Regulacje tego typu, o czym może nawet nie warto wspominać, powinny być poprzedzone dokładnym zapoznaniem się z dokumentacją kompilatora, starannym planowaniem i empirycznym sprawdzeniem, czy gra jest w ogóle warta świeczki.

Dostępy do pamięci

Niektóre procesory, zwłaszcza procesory sygnałowe, oferują rozkazy jednoczesnego przesyłania porcji danych z wielu źródeł do wielu celów, są to tak zwane multiple data moves. Można w ten sposób - jednym rozkazem - np. przesłać serię wartości całkowitych znajdujących się obok siebie w pamięci do kolejnych rejestrów. Kompilatory często wykorzystują tę możliwość dla maksymalizacji przepływu danych i tym samym lepszego wykorzystania wszystkich podsystemów procesora, jednostek obliczeniowych itp., które dzięki temu krócej czekają na dane do obróbki.

Istotnym szczegółem jest tu jednak wyrównanie danych w pamięci. Typowo kompilator wyrównuje zmienne do granicy ich rozmiaru: np. tablica zmiennych 16-bitowych zostanie umieszczona pod adresem podzielnym przez 2 (czyli parzystym). Ale żeby móc zastosować na niej przesył wielokrotny, dane muszą być wyrównane do granicy tej wielokrotności, np. jeśli chcemy załadować dwie wartości 16-bitowe naraz, pewnie będą się one musiały znaleźć na granicy czterech bajtów (tj. pod adresem podzielnym przez 4).

Wskaźniki

Kiedy dana funkcja realizuje dostępy (zwłaszcza zapisy) do pamięci przy użyciu wskaźników, dobrze jest się upewnić, że różne wskaźniki nie wskazują tego samego obszaru pamięci (czyli że się nie dublują lub, inaczej mówiąc, że jeden wskaźnik nie jest aliasem innego). Gdy kompilator jest pewien, że dowolny wskaźnik zawsze wskazuje inny obszar pamięci niż pozostałe wskaźniki, może zmienić kolejność dostępów tak jak mu wygodnie albo je zrównoleglić, a to znacznie zwiększa wydajność programu.

W przeciwnym wypadku, jeśli nie ma takiej pewności, kompilator musi założyć, że wskaźniki mogą wskazywać te same obszary pamięci, wobec tego zmiana kolejności lub paralelizacja dostępów jest znacznie utrudniona lub niemożliwa, gdyż mogłaby prowadzić do powstania nieprawidłowych wyników.

Trzeba zauważyć, że nie zawsze jest oczywiste, kiedy wskaźniki się dublują: może to zależeć od tego, co w danej chwili robi posługująca się nimi funkcja. Np. w przypadku takiej operacji:

```
memmove(s+256,s,256)
```

(gdzie s jest wskaźnikiem do obiektu typu char) wskaźniki źródłowy i docelowy się nie dublują, ale już przy memmove(s+256,s,257) jeden wskaźnik jest aliasem drugiego.

Do zasygnalizowania kompilatorowi, że dany wskaźnik nie dubluje się z żadnym innym, służy modyfikator restrict. Przez jego użycie programista gwarantuje, że w czasie istnienia wskaźnika (czyli w obrębie bloku programu, wewnątrz

którego ten wskaźnik zadeklarowano) dostęp do danego obiektu będzie realizowany wyłącznie za pośrednictwem tego wskaźnika. Większość kompilatorów pozwala to również zagwarantować globalnie dla wszystkich wskaźników. Niedotrzymanie tej gwarancji przez programistę spowoduje, że skutki działania programu będą wprawdzie trudne do przewidzenia, ale na pewno dalekie od zamierzonych.

Dyrektywa #pragma

Prawie każdy **kompilator języka C** oferuje możliwości unikalne dla danej implementacji. Można się takowymi posłużyć za pośrednictwem dyrektywy #pragma. Pozwala ona skorzystać ze specyficznych cech danej implementacji przy zachowaniu przez program ogólnej przenośności i zgodności z innymi implementacjami **języka**.

Przy użyciu tej dyrektywy można na przykład zasygnalizować kompilatorowi, że liczba iteracji danej pętli zawsze zawrze się w przedziale od konkretnego minimum do konkretnego maksimum. Gdy te dwa parametry są z góry znane, kod w tym miejscu może zostać lepiej zoptymalizowany; na przykład, jeśli minimum jest większe od zera, kompilator może zrezygnować z generowania dodatkowego kodu sprawdzającego, czy nie zachodzi sytuacja, kiedy pętlę należy w ogóle pominąć (bo ma się ona wykonać "zero razy"). Może mu to też pomóc w podjęciu decyzji, czy pętlę opłaca się rozwijać, a jeśli tak - to ile razy.

Pętle

Skoro już mowa o pętlach, nie od rzeczy będzie wspomnieć o tym, że pętle cechują się dodaniem do kodu pewnego narzutu wynikającego z konieczności wykonania dodatkowych rozkazów, które sterują przebiegiem programu. Takimi operacjami są typowo inkrementacja lub dekrementacja licznika pętli, sprawdzenie, czy osiągnął już stan docelowy oraz rozkaz skoku, który w zależności od stanu licznika może zostać wykonany albo nie. Z racji typowości tej konstrukcji znaczna część procesorów jest w stanie wykonać te trzy czynności - tj. modyfikację licznika, sprawdzenie stanu, skok - w ramach jednego rozkazu zajmującego mniej czasu niż trzy oddzielne. Pewien narzut nadal jednak istnieje.

Można się go pozbyć na dwa sposoby: tradycyjnym jest rozwinięcie pętli. Będzie o tym jeszcze mowa w dalszej części tekstu. Drugim, mniej tradycyjnym sposobem jest zastosowanie pętli sprzętowych, oferowanych przez niektóre architektury. Działa to w ten sposób, że mikroprocesor dostaje w rejestrach informację o początku i końcu pętli oraz o żądanej liczbie przebiegów. Ponieważ nie ma dodatkowych rozkazów sterujących przebiegiem programu, nie ma też generowanego przez nie narzutu.

Z racji braku narzutu, kompilatory na ogół dążą do generowania pętli sprzętowych, o ile mikroprocesor na to pozwala, nawet jeśli pętla jest bardzo skomplikowana. Niemniej prawie zawsze muszą zostać tu spełnione pewne kryteria, zależą one od danej architektury i implementacji kompilatora.

Znajomość tych kryteriów może - przez odpowiednie modyfikacje kodu źródłowego - pomóc programiście w spełnieniu ich przez program, co na ogół skutkuje powstaniem wydajniejszego kodu wynikowego. To zagadnienie może wymagać od programistów trochę wprawy i doświadczenia: niektóre kompilatory sygnalizują, że pętli sprzętowej nie udało się utworzyć, w przypadku innych trzeba będzie zajrzeć do wygenerowanego przez kompilator kodu w asemblerze.

Separacja danych

Jeśli mamy do czynienia z różnymi typami pamięci, trzeba mieć na uwadze, jak rozmieszczone są w nich dane. W zależności od rodzaju pamięci, jeśli dwie magistrale żądają dostępu do danego obszaru, banku itd., może powstać konfl ikt skutkujący opóźnieniem jednego z transferów. Żeby tego uniknąć, dane w pamięci trzeba rozmieszczać w sposób przemyślany - szczegóły tej czynności są zależne od danej architektury, bo konfiguracja pamięci i warunki dostępowe są różne w różnych urządzeniach.

Wyrównywanie danych w pamięci

Niektóre architektury nie robią formalnego rozróżnienia pomiędzy dostępem do danych, których położenie w pamięci wyrównano do pewnej granicy, a takimi bez tego wyrównania. Dlatego często możliwe jest np. załadowanie 32-bitowego słowa spod nieparzystego adresu, ale równie często tego typu operacja trwa dłużej niż załadowanie tego samego słowa spod adresu parzystego lub podzielnego przez cztery. Odpowiednie wyrównanie danych w pamięci może więc znacznie przyspieszyć dostęp do pamięci. W celu oceny, czy ten zabieg się opłaci na danej architekturze, trzeba oczywiście sięgnąć do dokumentacji.

Pamięć podręczna (cache)

Dane, które są obrabiane razem, opłaca się umieścić w pamięci tak, żeby znajdowały się jedne obok drugich i żeby już przy żądaniu dostępu do pierwszej z nich całość została załadowana do cache'u. Dodatkowo, na ile to możliwe, trzeba zadbać, żeby porcja danych obrabiana w jednym przebiegu pętli mieściła się całkowicie w jednej linii pamięci

cache. Dzięki temu dane znajdą się w cache'u tuż przed obróbką, a liczba przypadających na każdą iterację dostępów do zewnętrznej pamięci ulegnie znacznemu zmniejszeniu.

Funkcje inline

Normalną praktyką jest, że małe funkcje są wklejane przez kompilator wprost w miejsce wywołania. Jeśli natomiast z jakiegoś powodu nie dzieje się to automatycznie, bo, dajmy na to, włączona jest optymalizacja wielkości kodu raczej niż jego szybkości, programista może to wymusić przez zastosowanie modyfikatora inline lub odpowiedniej dyrektywy #pragma. Takie zabiegi zwiększają rozmiary programu, ale likwidują narzut związany z wywołaniem funkcji: zapisanie stanu rejestrów, odtworzenie ich, przekazanie parametrów itp., co w przypadku małych funkcji może być istotną częścią ich całkowitego czasu wykonania.

Użycie gotowych bibliotek

Producenci narzędzi deweloperskich dla systemów mikroprocesorowych dostarczają zwykle gotowe biblioteki zawierające typowe funkcje, wykonujące mniej lub bardziej złożone obliczenia w rodzaju FFT, FIR itd. Bardzo często te funkcje są napisane bezpośrednio w asemblerze i są szybsze niż ich odpowiedniki kompilowane w **języku C**, a już na pewno ich użycie skraca cykl powstawania programu o czas własnoręcznego ich napisania i przetestowania przez zespół tworzący daną aplikację. Zawsze warto z nich zatem skorzystać, o ile nie ma ku temu innych przeciwwskazań.

Przekształcenia pętli

Wspomniane nieco wyżej rozwijanie pętli polega na powieleniu jej głównej części (obliczeniowej) dwa lub więcej razy oraz na odpowiednim zmniejszeniu liczby iteracji. W przypadku zwykłej pętli zysk polega na zmniejszeniu procentowego udziału narzutu, generowanego przez rozkazy sterujące pętlą, w całkowitym czasie jej wykonania.

Ponadto daje to kompilatorowi więcej możliwości optymalizacji kodu, jako że każda kopia pętli ulega optymalizacji oddzielnie, a zatem kompilator może inaczej przydzielić rejestry, dokonać reorganizacji kodu na większą skalę (również pomiędzy kopiami) itp. W skrajnym przypadku pętla może zostać rozwinięta całkowicie, a jej część sterująca - zlikwidowana.

Jednak czy zabieg ten jest opłacalny, decyduje architektura konkretnego mikroprocesora: rozwijanie pętli zwiększa objętość kodu, w wyniku czego dana sekcja może się przestać mieścić w pamięci cache, a to - pociągnąć za sobą straty wydajności.

Paralelizm

Współczesne mikroprocesory dysponują wieloma, równolegle działającymi jednostkami przetwarzania danych, często również wieloma magistralami, co pozwala na jednoczesne wykonywanie wielu operacji, o ile tylko nie są od siebie bezpośrednio zależne. Kompilatory starają się do maksimum wykorzystać tę cechę mikroprocesorów, dokonując takiego przeorganizowania kodu, żeby jak najpełniej wykorzystać zasoby obliczeniowe sprzętu.

Jedną z technik, które im to ułatwiają, jest wspomniane wyżej rozwijanie pętli: pętla dokonująca obliczeń na tablicy danych przetwarza zwykle jedną pozycję tej tablicy (czyli jedną próbkę danych) w jednej iteracji. Gdy tę pętlę rozwiniemy N razy, jedna iteracja tym samym zacznie przetwarzać N próbek naraz, a to daje kompilatorowi okazję ku temu, żeby rozdzielić obliczenia dla kolejnych próbek pomiędzy wiele jednostek obliczeniowych i przetwarzać je równolegle. O tym, jak znacząco może to zwiększyć wydajność obliczeń w danym miejscu, nie trzeba nikogo przekonywać.

Możliwe są tu dwie drogi: zdanie się na kompilator, jak opisano powyżej, oraz jawne podzielenie zadania na wątki przy użyciu specjalnych bibliotek (np. Open MP). W tym drugim wypadku trzeba kompilatorowi nakazać podział danej sekcji na wątki konkretnymi dyrektywami #pragma.

Nie zawsze daje to dobre efekty, zwłaszcza jeśli program od początku powstawał jako jednowątkowy - liczba niejawnych zależności, uniemożliwiających jego efektywne przetwarzanie równoległe, może uczynić całą rzecz nieopłacalną. Dlatego, jeśli mamy do czynienia z procesorem wielordzeniowym, dobrze jest na samym wstępie poświęcić nieco czasu na zaplanowanie aplikacji - lub przynajmniej jej krytycznych części - od początku jako wielowątkowej.