|  |
| --- |
| BAN CƠ YẾU CHÍNH PHỦ  **HỌC VIỆN KỸ THUẬT MẬT MÃ** |
| ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯  Logo HvKTMM  **BÁO CÁO MÔN HỌC: TỐI ƯU PHẦN MỀM NHÚNG**  **ĐỀ TÀI:**  **TỐI ƯU HÓA VÒNG LẶP TRONG ASSEMBLY**  Ngành: Công nghệ thông tin  *Sinh viên thực hiện*:  **Mạc Văn Nam**  Lớp: CT2B  *Giảng viên hướng dẫn*:  **TS. Phạm Văn Hưởng**  Khoa Công nghệ thông tin – Học viện Kỹ thuật Mật mã  **Hà Nội, 2021** |

# LỜI CẢM ƠN

Trước tiên với tình cảm sâu sắc và chân thành nhất, cho phép em được bày tỏ lòng biết ơn đến các thầy cô ở khoa Công nghệ Thông tin – Học viện Kỹ thuật Mật mã đã truyền đạt vốn kiến thức quý báu cho em trong suốt thời gian học tập tại trường. Nhờ có những kiến thức mà thầy cô hướng dẫn, dạy bảo nên đề tài nghiên cứu củaem mới có thể hoàn thiện tốt đẹp.

Em xin gửi lời cảm ơn sâu sắc đến thầy Phạm Văn Hưởng – người đã trực tiếp giúp đỡ, quan tâm, hướng dẫn em hoàn thành tốt bài báo cáo này trong thời gian qua.

Vì thời gian và kiến thức còn hạn hẹp nên bài báo cáo không thể tránh khỏi những thiếu sót, em rất mong nhận được những ý kiến đóng góp quý báu của thầy để kiến thức củaem trong lĩnh vực này được hoàn thiện hơn đồng thời có cơ hội học thêm được nhiều kinh nghiệm và sẽ hoàn thành tốt hơn trong các bài báo cáo sắp tới.

Em xin chân thành cảm ơn!

# LỜI MỞ ĐẦU

Hệ thống nhúng là hệ thống máy tính cùng phần mềm được gắn theo một hệ thống khác để điều khiển hoạt động và xử lý thông tin của hệ thống đó. Khái niệm “phần mềm nhúng” được hiểu là phần mềm chạy trong các hệ thống nhúng.

Trong phát triển phần mềm nhúng, tối ưu là bài toán có ý nghĩa khoa học và mang tính thực tiễn cao. Tối ưu phần mềm nhúng có phạm vi nghiên cứu rộng và có thể được triển khai trong các giai đoạn khác nhau của vòng đời phần mềm. Bài toán tối ưu phần mềm nhúng cũng có thể được tiếp cận theo các mục tiêu tối ưu và các mức tối ưu khác nhau. Với mong muốn tìm hiểu, nghiên cứu về tối ưu hóa mã nguồn và những ứng dụng của nó, em đã lựa chọn đề tài “TỐI ƯU HÓA VÒNG LẶP TRONG ASSEMBLY” làm đề tài báo cáo môn học. Do thời gian kiến thức hạn hẹp nên đề tài chắc chắn không tránh khỏi những thiếu sót, vì thế em rất mong có được sự góp ý và nhắc nhở từ thầy để có thể hoàn thiện đề tài của mình.

Em xin chân thành cảm ơn thầy Phạm Văn Hưởng đã giúp đỡ em rất nhiều trong quá trình tìm hiểu, thực nghiệm và hoàn thành đề tài này.

# MỤC LỤC

[LỜI CẢM ƠN 1](#_Toc92143300)

[LỜI MỞ ĐẦU 2](#_Toc92143301)

[MỤC LỤC 2](#_Toc92143302)

[1 CHƯƠNG 1*:*TỐI ƯU HÓA VÒNG LẶP TRONG ASSEMBLY 5](#_Toc92143303)

[1.1 Minimize loop overhead 5](#_Toc92143304)

[1.2 Induction variable 7](#_Toc92143305)

[1.3 Move loop-invariant code 8](#_Toc92143306)

[1.4 Find the bottlenecks 8](#_Toc92143307)

[1.5 Instruction fetch, decoding and retirement in a loop 9](#_Toc92143308)

[1.6 Phân phối đồng đều μops giữa các đơn vị thực thi 10](#_Toc92143309)

[1.7 Loop unrolling 11](#_Toc92143310)

[1.8 Optimize caching 14](#_Toc92143311)

[1.9 Macro loop 16](#_Toc92143312)

[1.10 Song song hóa 19](#_Toc92143313)

[1.11 Vector loops using mask register (AVX512) 23](#_Toc92143314)

[2. CHƯƠNG 2*:* THỰC HÀNH TỐI ƯU HÓA MÃ NGUỒN C 28](#_Toc92143315)

[Tính toán các giá trị và đưa ra màn hình 28](#_Toc92143316)

[In ước chung lớn nhất của 2 số 29](#_Toc92143317)

[In ra giá trị các biến x,y,z 29](#_Toc92143318)

[Tính tổng n số bắt đầu từ m 29](#_Toc92143319)

[Tính trung bình cộng của dãy số 29](#_Toc92143320)

[In ra ký tự trong mảng 29](#_Toc92143321)

[Tính giai thừa 30](#_Toc92143322)

[Kiểm tra số nguyên tố 30](#_Toc92143323)

[Tính toán các biểu thức 30](#_Toc92143324)

[Tính toán các biểu thức với các số nguyên cho trước. 30](#_Toc92143325)

[3. CHƯƠNG 3*:*THỰC HÀNH TỐI ƯU HÓA HỢP NGỮ 32](#_Toc92143326)

[Tính toán các giá trị và đưa ra màn hình 32](#_Toc92143327)

[In ước chung lớn nhất của 2 số 32](#_Toc92143328)

[In ra giá trị các biến x,y,z 33](#_Toc92143329)

[Tính tổng n số bắt đầu từ m 34](#_Toc92143330)

[Tính trung bình cộng của dãy số 34](#_Toc92143331)

[In ra ký tự trong mảng 35](#_Toc92143332)

[Tính giai thừa 35](#_Toc92143333)

[Kiểm tra số nguyên tố 36](#_Toc92143334)

[Tính toán các biểu thức 36](#_Toc92143335)

[Tính toán các biểu thức với các số nguyên cho trước 37](#_Toc92143336)

[Tính toán các giá trị và đưa ra màn hình 38](#_Toc92143337)

[In ước chung lớn nhất của 2 số 38](#_Toc92143338)

[In ra giá trị các biến x,y,z 39](#_Toc92143339)

[Tính tổng n số bắt đầu từ m 39](#_Toc92143340)

[Tính trung bình cộng của dãy số 40](#_Toc92143341)

[In ra ký tự trong mảng 40](#_Toc92143342)

[Tính giai thừa 40](#_Toc92143343)

[Kiểm tra số nguyên tố 41](#_Toc92143344)

[Tính toán các biểu thức 42](#_Toc92143345)

[Tính toán các biểu thức với các số nguyên cho trước 43](#_Toc92143346)

[KẾT LUẬN 44](#_Toc92143347)

[TÀI LIỆU THAM KHẢO: 45](#_Toc92143348)

# CHƯƠNG 1*:*TỐI ƯU HÓA VÒNG LẶP TRONG ASSEMBLY

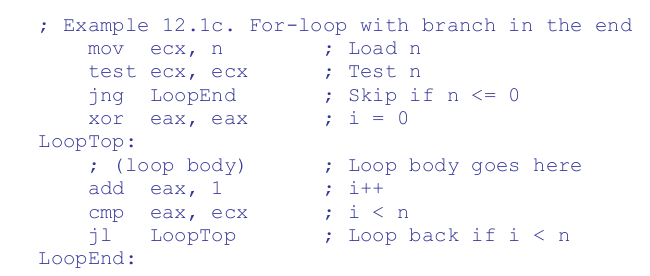
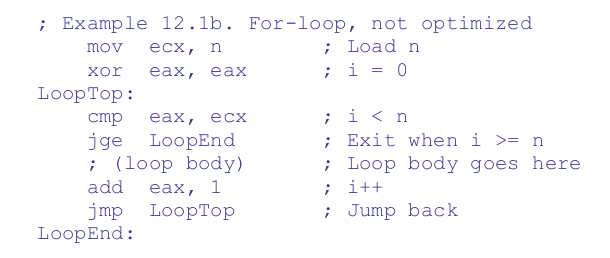
## Minimize loop overhead

Loop overhead là các lệnh sử dụng để quay trở lại phần đầu của vòng lặp và xác định thời điểm thoát khỏi vòng lặp. Tối ưu hóa các lệnh này là một kỹ thuật có thể được áp dụng trong nhiều tình huống. Tuy nhiên, tối ưu hóa loop overhead là không cần thiết, nếu chương trình đang bị hạn chế bởi các bottleneck.

Một số phương pháp để tối ưu các lệnh loop overhead:

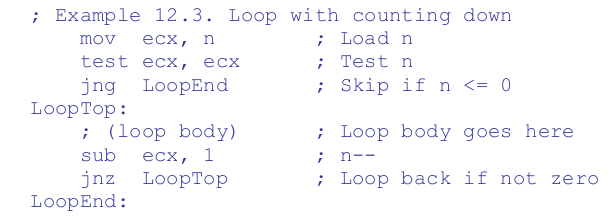
+ Cố gắng loại bỏ các lệnh jump trong vòng lặp, đặc biệt là các lệnh jump không cần đièu kiện (jmp) bằng cách xử dụng các exit branch ở cuối vòng lặp

vd : sử dụng exit branch jl thay vì lệnh jmp ở cuối vòng lặp



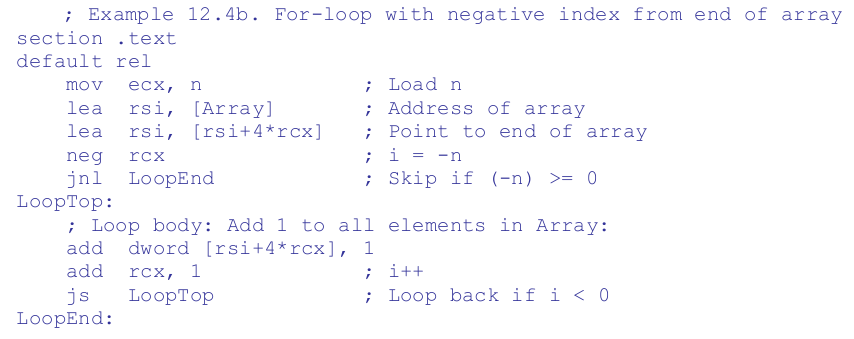
+ Sử dụng biến loop counter một cách hợp lý để hạn chế các lênh jump không điều kiện

vd: có thể loại bỏ lệnh nhảy ko điều kiện jmp và lệnh cmp so sánh giá trị của biến loop counter(n) bằng cách thay vì count từ 0→ n chuyển sang count ngược từ n→0



+ Đối với các vòng lặp có mảng trong vòng lặp: bộ đệm dữ liệu được tối ưu hóa để truy cập dữ liệu chuyển tiếp chứ không phải ngược lại nên cách đếm tốt nhất là đếm qua các giá trị âm từ -n đến 0, thực hiện được bằng cách tạo một con trỏ đến cuối mảng và sử dụng độ lệch âm(negative offset)(offset là độ dời của ô nhớ so với địa chỉ đầu của đoạn: segment) để đếm từ cuối mảng

vd:



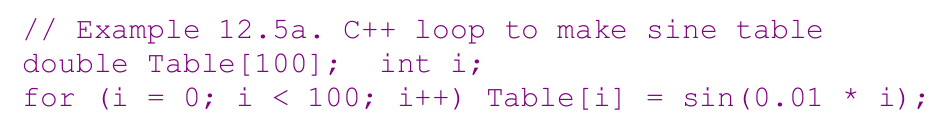
+ Biến loop counter nên là số nguyên thay vì là floating point vì lệnh so sánh floating point kém hiệu quả hơn so với so sánh số nguyên

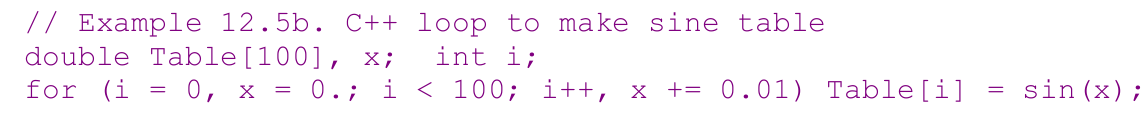
## Induction variable

Induction variable là biến có giá trị được tăng hoặc giảm cố định trong mỗi lần lặp lại của vòng lặp.

Nếu giá trị floating point bắt buộc phải sử dụng cho biến đếm của vòng lặp thì nên sử dụng kết hợp cả biến đếm nguyên và biến đếm floating point.

Vd: Sử dụng một bộ đếm số nguyên i cho điều khiển vòng lặp và chỉ số mảng, và một bộ đếm dấu phẩy động x để thay thế 0,01 \* i. Việc tính toán x bằng cách thêm 0,01 vào giá trị trước đó nhanh hơn nhiều so với việc chuyển i thành dấu phẩy động và nhân với 0,01





Ứng dụng : +Tính toán địa chỉ mảng

+Bất kỳ hàm nào là đa thức bậc n của biến đếm vòng lặp đều có thể được tính chỉ với n phép cộng và không có phép nhân bằng cách sử dụng n biến induction .

Việc sử dụng Induction variable có thể tạo nên các chuỗi phụ thuộc nên cần chú ý.

## Move loop-invariant code

Tính toán của bất kỳ biểu thức nào không thay đổi bên trong vòng lặp nên được chuyển ra ngoài vòng lặp.

Điều tương tự cũng áp dụng cho các nhánh if-else với điều kiện không thay đổi bên trong vòng lặp. Có thể tránh một nhánh như vậy bằng cách tạo hai vòng lặp, mỗi vòng một cho mỗi nhánh và tạo một nhánh chọn giữa hai vòng lặp.

## Find the bottlenecks

Một số bottlenecks xảy ra có thể hạn chế hiệu suất vòng lặp:

+ Cache miss :rạng thái nơi dữ liệu được yêu cầu xử lý bởi một thành phần không được tìm thấy trong bộ nhớ cache (cache memory).

+ Cache contentions: xảy ra khi tốc độ đọc > ghi , 2 hoặc nhiều CPU luân phiên cập nhật liên tục 1 dòng bộ nhớ cache

+ Chuỗi phụ thuộc xảy ra trong vòng lặp

+ Các lệnh fetching, decoding, retirement

+ Thông lượng cổng thực thi và thông lượng đơn vị thực thi

+ Sắp xếp lại thứ tự tối ưu và lập lịch của μops

+ Các nhánh dự đoán sai

## Instruction fetch, decoding and retirement in a loop

Nếu code fetching là một bottleneck thì cần phải căn chỉnh mục nhập vòng lặp bằng 16 và giảm kích thước lệnh để giảm thiểu số lượng ranh giới 16 byte trong vòng lặp.

Nếu lệnh decoding là một bottleneck thì cần phải tuân theo các quy tắc dành riêng cho CPU về các mẫu giải mã. Tránh các lệnh phức tạp tạo ra hơn hai μops, chẳng hạn như LOOP, JECXZ, LODS, STOS, v.v.

Nên tránh các bước nhảy và lệnh gọi bên trong vòng lặp vì nó làm chậm quá trình tìm nạp mã. Các chương trình con được gọi bên trong vòng lặp nên được nội tuyến nếu có thể.

Nên tránh các nhánh bên trong vòng lặp nếu có thể vì chúng cản trở việc dự đoán nhánh thoát vòng lặp. Tuy nhiên, không nên thay thế các nhánh bằng các động thái có điều kiện nếu điều này làm tăng độ dài của chuỗi phụ thuộc được thực hiện theo vòng lặp.

## Phân phối đồng đều μops giữa các đơn vị thực thi

Thời gian cần để gỡ bỏ tất cả các lệnh trong vòng lặp là tổng số μops chia cho tỷ lệ dừng. Tỷ lệ ngừng hoạt động là ít nhất 4 μops mỗi chu kỳ đồng hồ đối với bộ xử lý Intel Core2 trở lên. Thời gian retirement được tính toán là thời gian thực hiện tối thiểu cho vòng lặp. Giá trị này hữu ích như một tiêu chuẩn mà các tắc nghẽn tiềm ẩn khác có thể được so sánh với.

Thông lượng cho một cổng thực thi là 1 μop mỗi chu kỳ đồng hồ trên hầu hết các bộ xử lý Intel. Tải trên một cổng thực thi cụ thể được tính bằng số μops đi đến cổng này chia cho thông lượng của cổng. Nếu giá trị này vượt quá thời gian retirement như đã tính ở trên, thì cổng thực thi cụ thể này có khả năng là một bottleneck. Bộ xử lý AMD không có cổng thực thi, nhưng chúng có ba hoặc bốn đường ống với thông lượng tương tự.

Có thể có nhiều hơn một đơn vị thực thi trên mỗi cổng thực thi trên bộ xử lý Intel. Hầu hết các đơn vị thực thi có cùng thông lượng với cổng thực thi. Nếu đúng như vậy thì đơn vị thực thi không thể là một bottleneck. Nhưng một đơn vị thực thi có thể là một bottleneck trong các trường hợp sau: (1) nếu thông lượng của đơn vị thực thi thấp hơn thông lượng của cổng thực thi, ví dụ: để nhân và chia; (2) nếu đơn vị thực thi có thể truy cập được thông qua nhiều hơn một cổng thực thi; và (3) trên bộ xử lý AMD không có cổng thực thi.

Tải trên một đơn vị thực thi cụ thể được tính bằng tổng số μops đến đơn vị thực thi đó nhân với thông lượng tương hỗ cho đơn vị đó. Nếu giá trị này vượt quá thời gian retirement như đã tính ở trên, thì đơn vị thực thi cụ thể này có khả năng là một bottleneck.

## Loop unrolling

Một vòng lặp có n lần lặp lại có thể được thay thế bằng một vòng lặp lặp lại n / r lần và thực hiện r tính toán cho mỗi lần lặp lại, trong đó r là hệ số bỏ cuộn. n nên chia hết cho r.

Loop unrolling đc sử dụng để :

+ Giảm thiểu sử dụng loop overhead. loop overhead cho mỗi phép tính được giảm khi thực hiện chia cho hệ số bỏ cuộn vòng lặp r. Điều này chỉ hữu ích nếu loop overhead góp phần đáng kể vào thời gian tính toán. Không có lý do gì để bỏ cuộn một vòng lặp nếu một số bottleneck khác giới hạn tốc độ thực thi.

+Vectơ hóa. Một vòng lặp phải được rolled out bởi r hoặc bội số của r để sử dụng các thanh ghi vectơ có r phần tử.

+Cải thiện dự đoán của các exit branch. Dự đoán nhánh thoát vòng lặp có thể được cải thiện bằng cách mở vòng lặp đến mức số lần lặp lại n / r không vượt quá số lần lặp lại tối đa có thể được dự đoán trên một CPU cụ thể.

+Cải thiện bộ nhớ đệm. Nếu vòng lặp có nhiều lần bỏ sót bộ nhớ cache dữ liệu hoặc bộ nhớ cache nội dung thì có thể có lợi khi lên lịch đọc và ghi bộ nhớ theo cách tối ưu cho một bộ xử lý cụ thể. Điều này hiếm khi cần thiết trên các bộ vi xử lý hiện đại.

+Loại bỏ các phép chia số nguyên. Nếu vòng lặp chứa biểu thức trong đó bộ đếm vòng lặp i được chia cho số nguyên r, thì có thể tránh được phép chia số nguyên bằng cách bỏ vòng lặp cho r.

+Loại bỏ nhánh bên trong vòng lặp. Nếu có một nhánh hoặc một câu lệnh chuyển đổi bên trong vòng lặp với mẫu lặp lại của chu kỳ r thì điều này có thể được loại bỏ bằng cách bỏ cuộn vòng lặp theo r .

+Phá vỡ chuỗi phụ thuộc mang theo vòng lặp. Trong một số trường hợp, một chuỗi phụ thuộc được thực hiện theo vòng lặp có thể bị phá vỡ bằng cách sử dụng nhiều bộ tích lũy. Hệ số hủy cuộn r bằng số bộ tích lũy.

+Giảm sự phụ thuộc của induction variable. Nếu độ trễ của việc tính toán một biến cảm ứng từ giá trị trong lần lặp trước quá lâu đến mức nó trở thành bottleneck thì có thể giải quyết vấn đề này bằng cách mở cuộn theo r và tính toán từng giá trị của biến cảm ứng từ giá trị đó là r xếp sau trong dãy số.

+Hoàn tất việc hủy cuộn. Một vòng lặp hoàn toàn không được cuộn khi r = n, trong đó n là hằng số đã biết. Điều này giúp loại bỏ hoàn toàn chi phí vòng lặp. Mọi biểu thức là một hàm của bộ đếm vòng lặp đều có thể được thay thế bằng các hằng số. Mọi nhánh chỉ phụ thuộc vào bộ đếm vòng lặp đều có thể bị loại bỏ.

Cũng có những bất lợi đối với việc bỏ cuộn vòng lặp. Tính năng mở vòng lặp chỉ nên được sử dụng khi có lý do để làm như vậy và có thể đạt được tốc độ tăng đáng kể. Cần tránh việc bỏ cuộn vòng lặp quá mức. Nhược điểm của việc bỏ cuộn vòng lặp là:

• Mã trở nên lớn hơn và chiếm nhiều dung lượng hơn trong bộ đệm mã.

• Nhiều bộ xử lý có bộ đệm loopback để tăng tốc độ của các vòng lặp rất nhỏ. Bộ đệm lặp lại được giới hạn ở 20 - 50 lệnh hoặc 64 byte mã, tùy thuộc vào bộ xử lý. Hủy cuộn vòng lặp có thể làm giảm hiệu suất nếu nó vượt quá kích thước của bộ đệm lặp lại.

• Một số bộ xử lý có bộ nhớ đệm μop có kích thước hạn chế. Bộ đệm μop này rất có giá trị nên việc sử dụng nó phải được tiết kiệm. Các vòng lặp chưa được cuộn sẽ chiếm nhiều dung lượng hơn trong bộ đệm μop.

• Việc phải thực hiện thêm các phép tính bên ngoài vòng lặp không được thực hiện trong trường hợp n không chia hết cho r làm cho mã phức tạp hơn và làm tăng số nhánh, như giải thích bên dưới.

• Vòng lặp chưa được cuộn có thể cần nhiều thanh ghi hơn, ví dụ: cho nhiều bộ tích lũy.

Có một vấn đề cụ thể với việc bỏ cuộn vòng lặp khi số lặp lại n không chia hết cho hệ số bỏ cuộn r. Sẽ có một phần còn lại của n modulo r phép tính bổ sung không được thực hiện bên trong vòng lặp. Những tính toán bổ sung này phải được thực hiện trước hoặc sau vòng lặp chính.

Có thể khá khó khăn để thực hiện các phép tính phụ ngay khi số lặp lại không chia hết cho thừa số bỏ cuộn. Điều này đặc biệt phức tạp nếu chúng ta đang sử dụng một chỉ số âm như trong ví dụ 12,6c, d và e. Ví dụ sau đây hiển thị lại thuật toán DAXPY, lần này được giải phóng bằng 4, sử dụng AVX2. Trong ví dụ này, n là một biến có thể chia hết hoặc không chia hết cho 4.

Một giải pháp thay thế cho vòng lặp không cuộn thực hiện các phép tính trên mảng là mở rộng mảng với tối đa r-1 khoảng trống chưa sử dụng và làm tròn số lặp lại n thành bội số gần nhất của hệ số giải phóng r. Điều này loại bỏ nhu cầu tính toán phần còn lại (n modr) và đối với vòng lặp bổ sung cho các phép tính còn lại. Các phần tử mảng không sử dụng phải được khởi tạo bằng 0 hoặc một số giá trị dấu phẩy động hợp lệ khác để tránh các số ẩn hoặc bất kỳ điều kiện nào khác có thể làm chậm quá trình thả nổicác phép tính điểm. Nếu mảng có kiểu số nguyên thì điều kiện duy nhất bạn phải tránh chia cho 0.

## Optimize caching

Truy cập bộ nhớ có thể mất nhiều thời gian hơn trong một vòng lặp mà việc truy cập bộ nhớ chưa được lưu trữ. Dữ liệu phải được giữ liền kề nếu có thể và được truy cập tuần tự

Số mảng được truy cập trong một vòng lặp không được vượt quá số bộ đệm đọc / ghi trong bộ vi xử lý. Một cách để giảm số lượng luồng dữ liệu là kết hợp nhiều mảng thành một mảng cấu trúc để nhiều luồng dữ liệu được xen kẽ vào một luồng duy nhất.

Các bộ vi xử lý hiện đại có cơ chế tìm nạp trước dữ liệu tiên tiến. Các cơ chế này có thể phát hiện các quy định trong mẫu truy cập dữ liệu, chẳng hạn như truy cập dữ liệu với một bước ghi 100 cụ thể. Bạn nên tận dụng các cơ chế tìm nạp trước như vậy bằng cách giữ số lượng luồng dữ liệu khác nhau ở mức tối thiểu và giữ tốc độ truy cập không đổi nếu có thể. Tìm nạp trước dữ liệu tự động thường hoạt động tốt hơn tìm nạp trước dữ liệu rõ ràng khi mẫu truy cập dữ liệu đủ thường xuyên.

Tìm nạp trước dữ liệu rõ ràng với các hướng dẫn tìm nạp trước có thể cần thiết trong trường hợp mẫu truy cập dữ liệu quá bất thường để được dự đoán tự động. Thường cần nhiều thử nghiệm để tìm ra chiến lược tìm nạp trước tối ưu cho một chương trình truy cập dữ liệu theo cách không thường xuyên.

Có thể đưa dữ liệu tìm nạp trước vào một luồng riêng biệt nếu bộ vi xử lý có thể chạy hai luồng trong mỗi lõi CPU.

Các hướng dẫn ghi không theo thời gian rất hữu ích để ghi vào bộ nhớ chưa được xóa mà không có khả năng được truy cập lại sớm. Bạn có thể sử dụng hướng dẫn vectơ để giảm thiểu số lượng lệnh viết không theo thời gian.

Cách quan trọng nhất để cải thiện hiệu suất của mã sử dụng CPU là thực hiện song song mọi thứ. Các phương pháp chính để thực hiện song song là:

• Cải thiện khả năng thực thi không đúng trình tự của CPU. Điều này được thực hiện bằng cách phá vỡ các chuỗi phụ thuộc dài (xem trang 64) và phân phối đồng đều μops giữa các đơn vị thực thi hoặc cổng thực thi khác nhau

• Sử dụng hướng dẫn vectơ

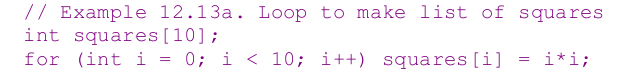
Chuỗi phụ thuộc được thực hiện theo vòng lặp có thể bị phá vỡ bằng cách sử dụng nhiều bộ tích lũy, như được giải thích trên trang 64. Số bộ tích lũy hữu ích tối đa là độ trễ của lệnh quan trọng nhất trong chuỗi phụ thuộc chia cho thông lượng tương hỗ của lệnh đó. Ví dụ: nếu độ trễ của phép cộng dấu phẩy động là 4 chu kỳ đồng hồ và thông lượng tương hỗ là 1, thì số bộ tích lũy hữu ích tối đa là 4. Số lượng bộ tích lũy thấp hơn mức tối đa có thể đủ để đảm bảo sự phụ thuộc vào vòng lặp chuỗi không phải là một yếu tố hạn chế.

## Macro loop

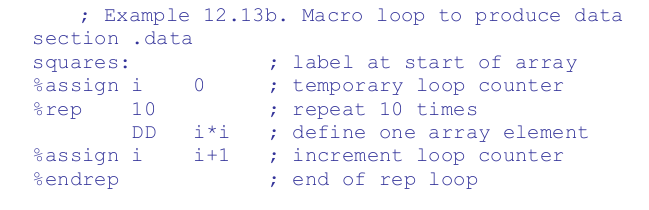
Nếu số lần lặp lại của một vòng lặp nhỏ và không đổi, thì có thể hủy cuộn hoàn toàn vòng lặp. Ưu điểm của điều này là các phép tính chỉ phụ thuộc vào bộ đếm vòng lặp có thể được thực hiện tại thời điểm lắp ráp hơn là tại thời điểm thực thi. Tất nhiên, nhược điểm là nó chiếm nhiều dung lượng hơn trong bộ đệm mã nếu số lần lặp lại cao.

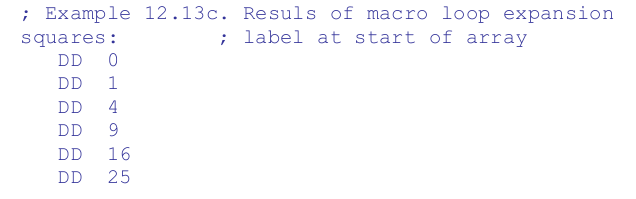
Cú pháp NASM bao gồm một ngôn ngữ macro, có thể khá hữu ích. Các trình lắp ráp khác có khả năng tương tự, nhưng cú pháp thì khác.

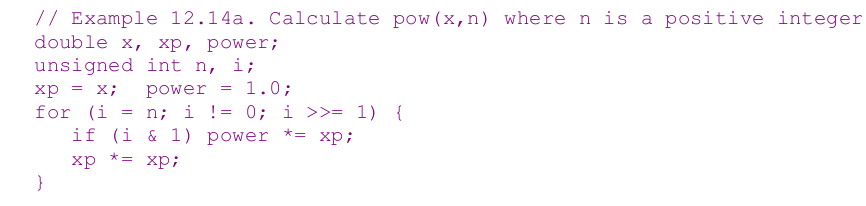
Ví dụ: nếu chúng ta cần một danh sách các số bình phương, thì mã C ++ có thể trông giống như sau:

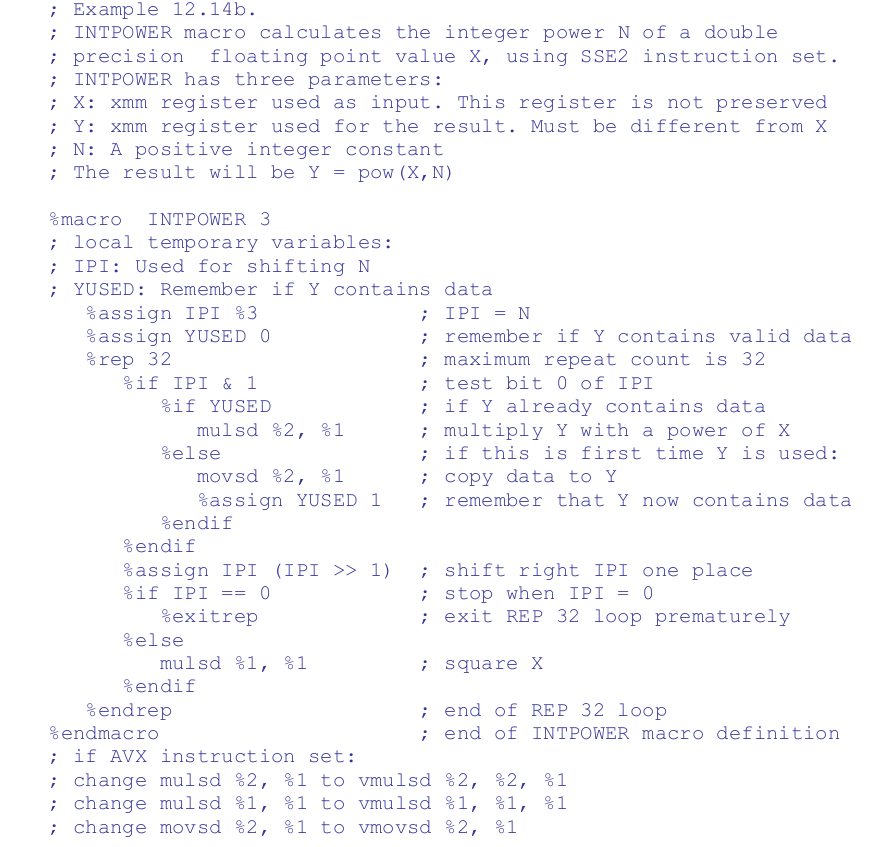


Cùng một danh sách có thể được tạo bằng một vòng lặp macro trong ngôn ngữ NASM:

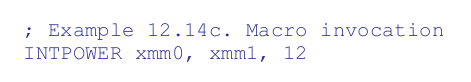
Ở đây, i là một biến tiền xử lý. Vòng lặp i được chạy tại thời điểm lắp ráp, không phải tại thời điểm thực thi. Biến i và câu lệnh% gán i i + 1 không bao giờ đưa nó vào mã cuối cùng và do đó không mất thời gian để thực thi. Trên thực tế, ví dụ 12.13b không tạo ra mã thực thi mà chỉ có dữ liệu. Bộ xử lý tiền macro sẽ dịch đoạn mã trên thành:

Các vòng lặp macro cũng hữu ích để tạo mã. Ví dụ tiếp theo tính x n, trong đó x là số dấu phẩy động và n là số nguyên dương. Điều này được thực hiện hiệu quả nhất bằng cách bình phương nhiều lần x và nhân với nhau các thừa số tương ứng với các chữ số nhị phân trong n. Thuật toán có thể được thể hiện bằng mã C ++:

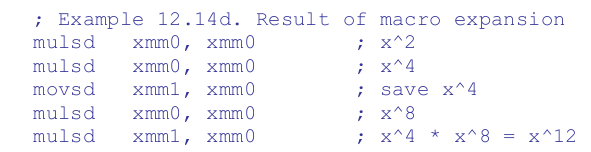
Nếu n được biết tại thời điểm lắp ráp, thì hàm nguồn có thể được thực hiện bằng cách sử dụng vòng lặp macro sau:



Macro này tạo ra số lượng lệnh tối thiểu cần thiết để thực hiện công việc. Không có chi phí vòng lặp, đoạn mở đầu hoặc phần epilog trong mã cuối cùng. Và, quan trọng nhất, không có chi nhánh. Tất cả các nhánh đã được giải quyết bởi bộ tiền xử lý macro. Để tính xmm0 theo lũy thừa của 12, bạn viết:



Điều này sẽ được mở rộng thành:



Điều này thậm chí có ít hướng dẫn hơn so với một vòng lặp lắp ráp được tối ưu hóa mà không cần cuộn. Macro cũng có thể hoạt động trên vectơ khi mulsd được thay thế bằng mulpd và movsd được thay thế bằng movapd.

## Song song hóa

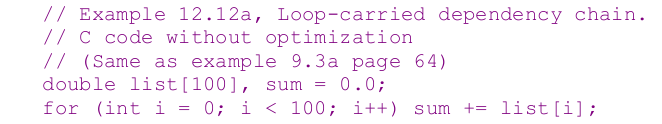
Cách quan trọng nhất để cải thiện hiệu suất của mã sử dụng CPU là thực hiện song song mọi thứ. Các phương pháp chính để thực hiện song song là:

• Cải thiện khả năng thực thi không đúng trình tự của CPU. Điều này được thực hiện bằng cách phá vỡ các chuỗi phụ thuộc dài (xem trang 64) và phân phối đồng đều μops giữa các đơn vị thực thi hoặc cổng thực thi khác nhau (xem trang 92).

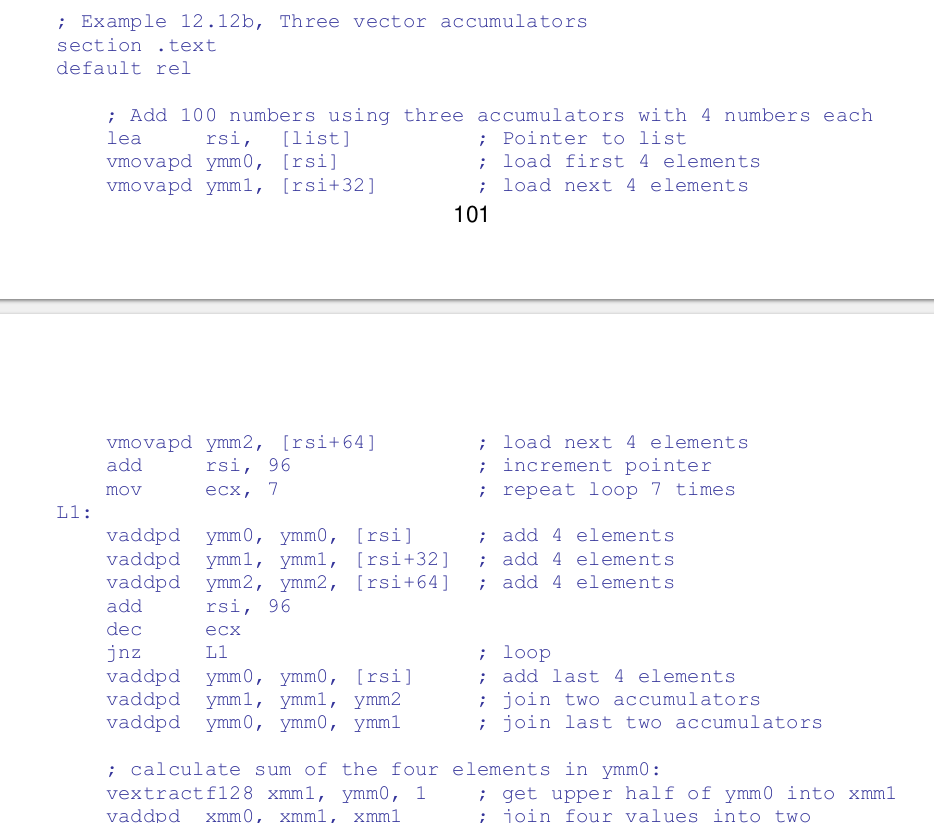
• Sử dụng vectơ instruction.

• Sử dụng multi thread.

Chuỗi phụ thuộc được thực hiện theo vòng lặp có thể bị phá vỡ bằng cách sử dụng nhiều bộ tích lũy, như được giải thích trên trang 64. Số bộ tích lũy hữu ích tối đa là độ trễ của lệnh quan trọng nhất trong chuỗi phụ thuộc chia cho thông lượng tương hỗ của lệnh đó. Ví dụ: nếu độ trễ của phép cộng dấu phẩy động là 4 chu kỳ đồng hồ và thông lượng tương hỗ là 1, thì số bộ tích lũy hữu ích tối đa là 4. Số lượng bộ tích lũy thấp hơn mức tối đa có thể đủ để đảm bảo sự phụ thuộc vào vòng lặp chuỗi không phải là một yếu tố hạn chế.



Ví dụ 12.12b cho thấy một vòng lặp có thêm một trăm số, sử dụng ba thanh ghi vectơ AVX2 làm bộ tích lũy.



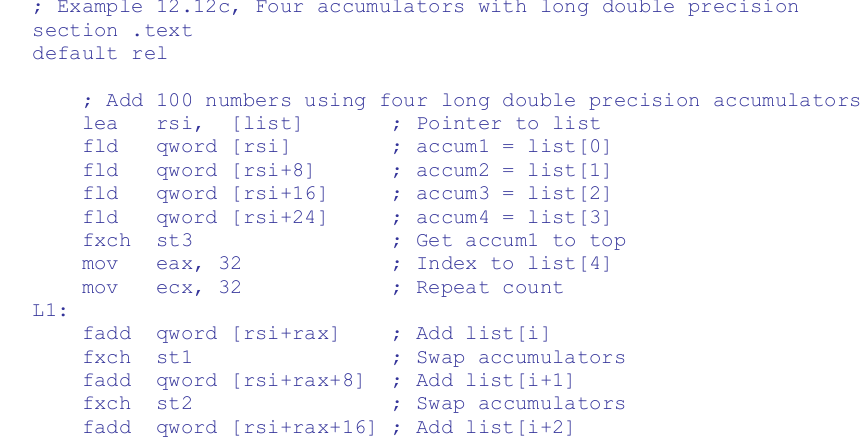
Trong ví dụ 12.12b, tôi đã tải ba thanh ghi vectơ có giá trị 3 \* 4 đầu tiên từ danh sách. Vòng lặp thực hiện 12 lần bổ sung cho mỗi lần lặp. Chúng ta cần thực hiện thêm bốn phép cộng nữa sau vòng lặp vì 100 không chia hết cho 12. Ba tích chứa bốn tổng từng phần được cộng lại với nhau để tạo thành một vectơ gồm bốn giá trị sau vòng lặp. Cuối cùng, bốn giá trị này được cộng lại với nhau để có được tổng cuối cùng.

Vòng lặp trong ví dụ 12.12b có bốn chuỗi phụ thuộc được thực hiện theo vòng lặp chạy song song:

ymm0, ymm1, ymm2 và ecx. Tốc độ bị giới hạn bởi độ trễ của lệnh vaddpd, lâu hơn độ trễ của lệnh dec ecx. Có thể cải thiện tốc độ hơn nữa bằng cách sử dụng nhiều bộ tích lũy hơn, nhưng các tác động khác có khả năng làm chậm quá trình thực thi khi chúng ta đạt đến thông lượng giới hạn của bộ cộng dấu phẩy động. Số lượng bộ tích lũy tối ưu thường ít hơn số lượng tối đa lý thuyết. Với chỉ 7 lần lặp lại của vòng lặp trong ví dụ này, có lẽ không đáng để thêm mã để thêm nhiều bộ tích lũy.

Không cần sử dụng các chỉ mục âm trong vòng lặp này vì chi phí vòng lặp không phải là một nút cổ chai giới hạn.

Tình huống trở nên phức tạp hơn nếu bạn muốn sử dụng độ chính xác kép dài để giảm sự tích tụ của lỗi làm tròn. Độ chính xác kép dài chỉ khả dụng với các thanh ghi dấu chấm động kiểu x87 cũ. Các thanh ghi này được tổ chức như một ngăn xếp cuộn. Chúng ta phải sử dụng hướng dẫn fxch để đưa đăng ký mong muốn lên đầu ngăn xếp.

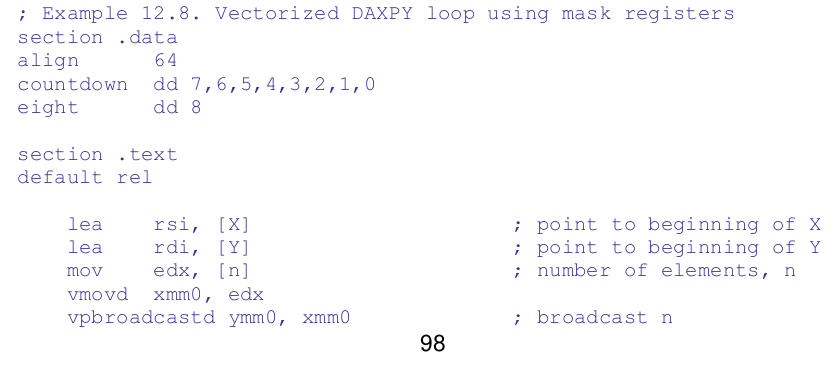


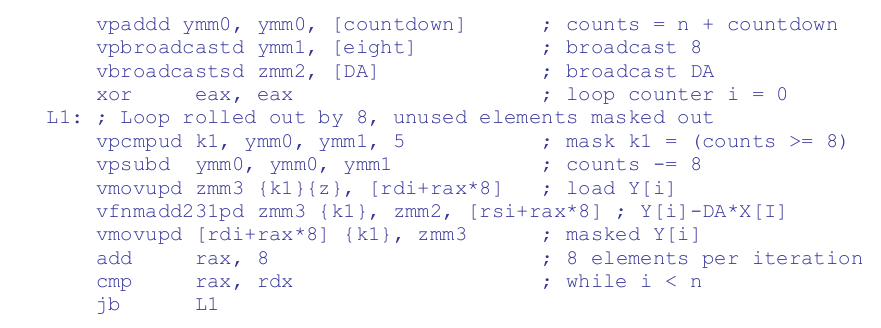
Trong ví dụ 12.12c, chúng tôi đã tải bốn bộ tích lũy với bốn giá trị đầu tiên từ danh sách. Điều thú vị khi sử dụng thanh ghi x87 làm bộ tích lũy là số bộ tích lũy bằng hệ số triển khai cộng với một. Đây là hệ quả của cách các hướng dẫn fxch được sử dụng để hoán đổi các bộ tích lũy. Bạn phải chơi máy tính và theo dõi vị trí của mỗi bộ tích lũy trên ngăn xếp thanh ghi dấu phẩy động để xác minh rằng bốn bộ tích lũy thực sự được xoay một nơi sau mỗi lần lặp lại của vòng lặp để mỗi

bộ tích lũy được sử dụng cho mỗi lần bổ sung thứ tư mặc dù thực tế là vòng lặp chỉ được triển khai bởi ba.

## Vector loops using mask register (AVX512)

Tập lệnh AVX512 cung cấp các hoạt động được che giấu. Tính năng này hữu ích để che đi các phần tử vectơ thừa khi số vòng lặp không chia hết cho kích thước vectơ:



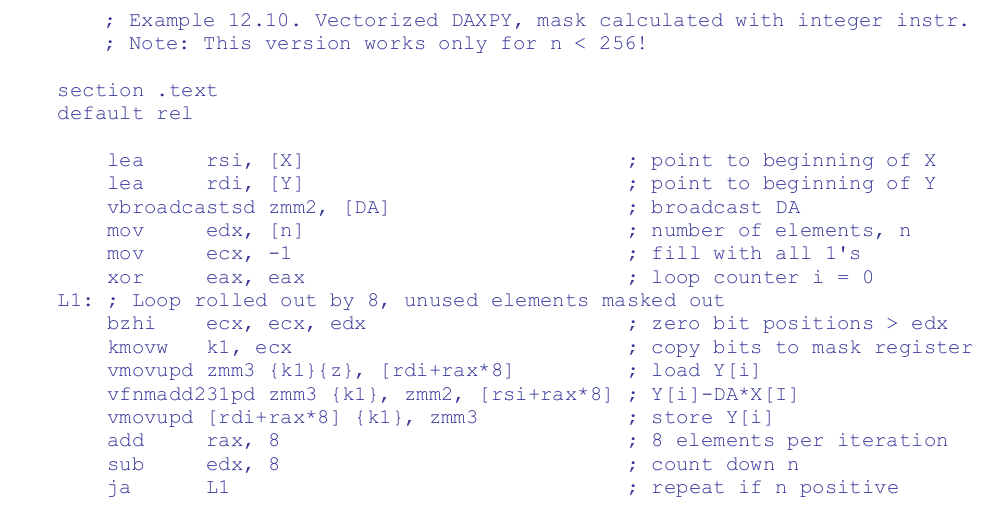


Ở đây, mặt nạ trong thanh ghi k1 có bit 1 cho tất cả các phần tử hợp lệ và 0 cho các phần tử thừa ngoài n. Số phần tử còn lại được đếm ngược trong YMM0.

Các hướng dẫn tải và lưu trữ được che bởi k1 để tránh tải và lưu trữ bất kỳ thứ gì ngoài n phần tử của Y. Lệnh tải bị che có tùy chọn zeroing {z} để tránh phụ thuộc sai vào giá trị trước đó của thanh ghi vectơ ZMM3. Tốt là nên che các hướng dẫn tính toán bằng cách sử dụng cùng một mặt nạ. Việc che dấu các phép tính giúp tiết kiệm điện năng và tránh các ngoại lệ và hình phạt đối với các giá trị siêu thường có thể có, v.v. Không có chi phí để thêm mặt nạ vào lệnh vectơ 512-bit.

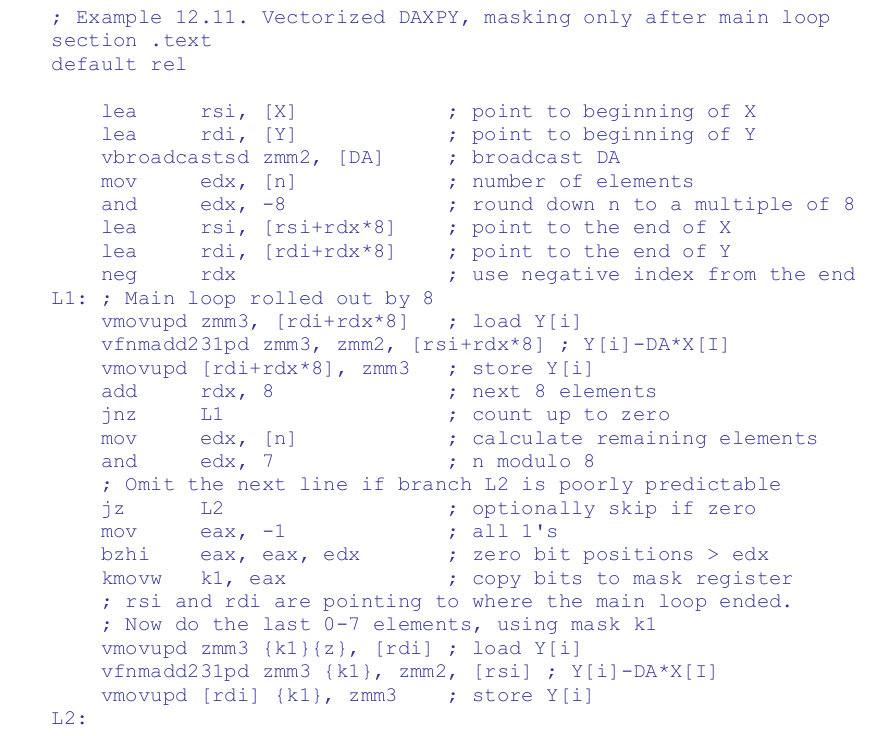
Các lệnh đếm ngược được sử dụng để tạo mặt nạ phải có cùng số phần tử trên mỗi vectơ như các phép tính, nhưng không nhất thiết phải có cùng số bit trên mỗi phần tử. Chúng tôi đang đếm ngược n trong YMM0 với tám số nguyên không dấu 32 bit, trong khi các phép tính DAXPY đang sử dụng tám phao chính xác kép 64 bit trong ZMM3.

Vòng lặp trong ví dụ 12.9 bị chi phối bởi các lệnh vectơ. Có khả năng các đơn vị thực thi vector sẽ là một nút thắt cổ chai ở đây. Có thể tốt hơn nếu sử dụng hướng dẫn số nguyên thay vì hướng dẫn vectơ để tạo mặt nạ nếu các đơn vị thực thi cho phép cộng vectơ là một nút cổ chai. Điều này được hiển thị trong ví dụ tiếp theo:



Ví dụ 12.10 đếm ngược số phần tử còn lại trong edx và sử dụng bzhi để xóa các bit còn lại trong ecx trong lần lặp cuối cùng nếu edx <16. Lệnh bzhi thuộc tập lệnh BMI2, được hỗ trợ trên tất cả các bộ xử lý đã biết với AVX512. Lưu ý rằng điều này chỉ hoạt động với n <256 vì bzhi chỉ đọc 8 bit thấp hơn của edx. Chúng tôi đang sử dụng lệnh kmovw thay vì kmovb vì lệnh thứ hai yêu cầu tập lệnh AVX512DQ.

Nếu số lần lặp cao và hướng dẫn tính mặt nạ làm chậm quá trình thực thi vòng lặp thì tốt hơn nên có một vòng lặp chính không có mặt nạ và chỉ thực hiện các hoạt động yêu cầu mặt nạ sau vòng lặp chính:



Ví dụ 12.11 chứa phần thân của vòng lặp hai lần. Phần thân thứ nhất được lặp lại n / 8 lần trong vòng lặp L1. Phần thân thứ hai tính n phần tử còn lại theo modulo 8 khi n không chia hết cho 8. Không nhất thiết phải bỏ qua phần thân cuối với jz L2 nếu nhánh này kém dự đoán.

Ví dụ 12.11 chỉ có 5 lệnh trong thân vòng lặp. Đây sẽ là giải pháp nhanh nhất nếu số vòng lặp cao. Ví dụ 12.8 và 12.10 rất hữu ích nếu phần thân của vòng lặp lớn đến mức chúng ta không muốn gộp nó hai lần hoặc nếu số vòng lặp nhỏ, hoặc nếu nút cổ chai là quyền truy cập bộ nhớ hoặc một chuỗi phụ thuộc được thực hiện theo vòng lặp. Ví dụ 12.10 chỉ có thể được sử dụng nếu n <256.

# CHƯƠNG 2*:* THỰC HÀNH TỐI ƯU HÓA MÃ NGUỒN C

Có nhiều phương pháp tối ưu mã nguồn trong một chương trình C, ví dụ như tối ưu hóa vòng lặp, tinh giản biểu thức toán học, tối ưu biểu thức điều kiện, giảm các biểu thức lặp, sử dụng cờ tối ưu hóa GCC, ... Các phương pháp này đều hướng đến những mục đích giảm kích thước mã nguồn và tăng hiệu năng của chương trình.

Bảng 1 dưới đây mô tả các ví dụ chương trình, phương pháp tối ưu và những kết quả đạt được sau khi tối ưu:

*Bảng 1:*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Mô tả chương trình** | **Kỹ thuật tối ưu thực hiện** | **Mô tả kỹ thuật** | **Thời gian thực hiện/ Kích thước** | | |
| **Chương trình gốc** | **Chương trình tối ưu** | **% cải tiến** |
| 1 | Tính toán các giá trị và đưa ra màn hình | Tính trung bình cộng và in ra các số chia hết cho 5 trong dãy số. | Thay đổi thứ tự các đoạn mã trong chương trình. | Hoán đổi vị trí mã tính trung bình cộng và mã in ra số chia hết cho 5. | 0.000058s | 0.000040s | 31% |
| 2 | In ước chung lớn nhất của 2 số | In ra ước chung lớn nhất của 2 số a và b | Thay đổi thuật toán tính ước chung lớn nhất | Thay đổi thuật toán tính ước chung lớn nhất | 0.001704s | 0.000021s | 2.15% |
| 3 | In ra giá trị các biến x,y,z | Tính giá trị chuỗi đầu vào và in ra giá trị các biến đã cho | Sử dụng biến thay thế | Đặt biến len bằng độ dài của chuỗi để giảm việc tính toán khi phải gọi hàm strlen | 0.000078 | 0.000057 | 27% |
| 4 | Tính tổng n số bắt đầu từ m | Tính tổng n số bắt đầu từ m. | Thay đổi thuật toán | Thay thế vòng lặp bằng cách sử dụng công thức toán học để tính tổng. | 0.00009s | 0.00008s | 11.11% |
| 5 | Tính trung bình cộng của dãy số | Tính trung bình cộng của dãy số từ 1 đến 1000000. | Sử dụng kỹ thuật unroll loop | Tăng biến đếm lên 4 để giảm số lần lặp. | 0.023178s | 0.007621s | 67.1% |
| 6 | In ra ký tự trong mảng | In ra ký tự trong mảng | Sử dụng mảng tối ưu | Thay thế mảng 2 chiều bằng mảng 1 chiều | 0.000066s | 0.000063 (-O1)  0.000057s (-O2)  0.000051 (-O3) | 4.5%  13%  22.7% |
| 7 | Tính giai thừa | Tính giai thừa từ 1 tới n | Sử dụng toán tử đơn giản | Thay thế toán tử <= bằng toán tử != | 0.000069s | 0.000053s | 23% |
| 8 | Kiểm tra số nguyên tố | Kiểm tra xem số n đầu vào có phải số nguyên tố không | Unroll Loop | Thay đổi giá trị điều kiện kết thúc từ n-1 thành sqrt(n) | 0.00009s | 0.000045s | 50% |
| 9 | Tính toán các biểu thức | Tính toán phép tính của 4 số cho trước . | Tối ưu hóa vòng lặp | Đưa các biểu thức có thể tính toán ngoài vòng lặp ra khỏi vòng lặp.Tăng tốc độ xử lý | 0.00008s | 0.00004s | 50% |
| 10 | Tính toán các biểu thức với các số nguyên cho trước. | Tính tổng từ 1 đến n, tổng giai thừa 1! đến n! và một số biểu thức liên quan. | Di chuyển mã để giảm kích thước; loại bỏ mã chết | Di chuyển mã ra khỏi lệnh if else, thực hiện loại bỏ mã chết tìm tổng phân số, do đoạn mã này không liên quan đến yêu cầu chương trình. | 984byte | 854byte | 13.2% |

# CHƯƠNG 3*:*THỰC HÀNH TỐI ƯU HÓA HỢP NGỮ

Có một số phương pháp làm tăng hiệu năng biên dịch hay giảm kích thước file thực thi trong quá trình tối ưu hóa mã hợp ngữ, có thể kể đến như: tối ưu hóa vòng lặp, loại bỏ mã chết, biến đổi các lệnh phức tạp hay sắp xếp mã nguồn, ...

Bảng 3 và 4 mô tả các chương trình C được compile thành hợp ngữ trong kiến trúc ARM và X86, phương pháp tối ưu và những kết quả đạt được sau khi tối ưu.

*Bảng 2: Tối ưu hoá X86*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Mô tả chương trình** | **Kỹ thuật tối ưu thực hiện** | **Mô tả kỹ thuật** | **Thời gian thực hiện/ Kích thước** | | |
| **Chương trình gốc** | **Chương trình tối ưu** | **% cải tiến** |
| 1 | Tính toán các giá trị và đưa ra màn hình | Tính trung bình cộng và in ra các số chia hết cho 5 trong dãy số. | Giảm số lần lặp | Tăng biến đếm lên 5 để giảm số lần lặp | 0.000067s | 0.000051s | 23% |
| Loại bỏ mã chết | Loại bỏ phần mã thực hiện việc xét xem i chia hết cho 5 không | 0.000061 | 0.000056 | 8% |
| 2 | In ước chung lớn nhất của 2 số | In ra ước chung lớn nhất của 2 số a và b | Thay thế các lệnh phức tạp bằng các lệnh tương đương | Tránh sử dụng cú pháp MOV $0, %EAX. Cú pháp này được thay thế bằng lệnh XORL %EAX, %EAX  Lệnh LEAVE sao chép EBP vào ESP và sau đó khôi phục lại EBP từ ngăn xếp. Lệnh này có thể thay thế thành cặp lệnh MOVQ-POPQ | 0.000022s | 0.000020s | 9% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Lệnh LEAVE sao chép EBP vào ESP và sau đó khôi phục lại EBP từ ngăn xếp. Lệnh này có thể thay thế thành cặp lệnh MOVQ-POPQ  Tránh sử dụng cú pháp MOV $0, %EAX. Cú pháp này được thay thế bằng lệnh XORL %EAX, %EAX | 0.000017s | 0.000016 | 5% |
| 3 | In ra giá trị các biến x,y,z | Tính giá trị chuỗi đầu vào và in ra giá trị các biến đã cho | Sử dụng biến thay thế | Đặt -24(%rbp) làm biến chung để tính toán | 0.000071s | 0.000068s | 4.2% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thây thế lệnh inc -28(%rbp) bằng lệnh addl $1, -28(%rbp) | 0.000054s | 0.000050s | 7.4% |
| 4 | Tính tổng n số bắt đầu từ m | Tính tổng n số bắt đầu từ m | Sử dụng biến thay thế | Sử dụng -20(%rbp) để lưu giá trị tổng của -24(%rbp) và -28(%rbp) để tránh việc tính toán trong vòng lặp | 0.00006s | 0.00006s |  |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | MOV $0, %EAX được thay thế bằng lệnh XORL %EAX, %EAX  Lệnh LEAVE sao chép EBP vào ESP và sau đó khôi phục lại EBP từ ngăn xếp. Lệnh này có thể thay thế thành cặp lệnh MOVQ-POPQ  Tránh sử dụng cú pháp MOV $1, %EAX. Cú pháp này được thay thế bằng lệnh XORL %EAX, %EAX / INC %EAX | 0.00006s | 0.00006s |  |
| 5 | Tính trung bình cộng của dãy số | Tính trung bình cộng của dãy số từ 1 đến 1000000 | Sử dụng kỹ thuật unroll loop | Tăng biến đếm -20(%RBP) lên 4 để giảm số lần lặp. | 0.023404s | 0.023233s | 0.7% |
| Sử dụng thanh ghi thay thế | Không so sánh trực tiếp thanh rbp vs eax mà dùng thanh edx thay thế | 0.008064s | 0.007654s | 5% |
| 6 | In ra ký tự trong mảng | In ra ký tự trong mảng | Tối ưu kích thước bằng cờ -Os | Sử dụng cờ Os để tối ưu kích thước | 0.000065s | 0.000065s | 0% |
| Tối ưu kích thước bằng cờ -Os | Sử dụng cờ Os để tối ưu kích thước | 0.000059s | 0.000059s | 0% |
| 7 | Tính giai thừa | Tính giai thừa từ 1 tới n | Thay thế các lệnh phức tạp bằng các lệnh tương đương | thay lệnh jle bằng jne | 0.000058s | 0.000054s | 6.8% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | thay lệnh decl bằng subl | 0.000054s | 0.000050s | 7.4% |
| 8 | Kiểm tra số nguyên tố | Kiểm tra xem số n đầu vào có phải số nguyên tố không | Unroll Loop | Thay đổi giá trị điều kiện kết thúc từ n-1 thành sqrt(n) | 0.000062s | 0.000055s | 11% |
| Sử dụng biến thay thế | đặt biến phụ có giá trị bằng sqrtn | 0.000053s | 0.000046s | 13% |
| 9 | Tính toán các biểu thức | Tính toán phép tính của 4 số cho trước . | Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế các câu lệnh như MOV $1, %EAX, MOV $0, %EAX, LEAVE | 0.00008s | 0.00007s | 12.5% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế các câu lệnh như XORPS %XMM0, %XMM0, MOV $1, %EAX, MOV $0, %EAX, LEAVE | 0.00004s | 0.00003s | 25% |
| 10 | Tính toán các biểu thức với các số nguyên cho trước | Tính tổng từ 1 đến n, tổng giai thừa 1! đến n! và một số biểu thức liên quan. | Loại bỏ mã chết | Loại bỏ mã tổng phân số không cần thiết | 0.000067s | 0.000054s | 19.4% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Sử dụng cờ -O3 để thay thế thanh ghi và thay thế các lệnh phức tạp thành các lệnh đơn giản tương đương và thay thế câu lệnh MOVL $1 | 0.000065s | 0.000054s | 16.9% |

*Bảng 3: Tối ưu hoá ARM*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Mô tả chương trình** | **Kỹ thuật tối ưu thực hiện** | **Mô tả kỹ thuật** | **Thời gian thực hiện/ Kích thước** | | |
| **Chương trình gốc** | **Chương trình tối ưu** | **% cải tiến** |
| 1 | Tính toán các giá trị và đưa ra màn hình | Tính trung bình cộng và in ra các số chia hết cho 5 trong dãy số. | Unroll loop | adds r3, r3, #5 ; tăng biens đếm lên 5 để giảm vòng lặp | 0.002408s | 0.002376s | 1.32% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | movs r3, #0 -> ands r3, #0  movs r3, #1 -> ands r3, #0/ add r3, # | 0.002383 | 0.002240 | 1.8% |
| 2 | In ước chung lớn nhất của 2 số | In ra ước chung lớn nhất của 2 số a và b | Thay thế các lệnh phức tạp bằng các lệnh tương đương | movs r3, #0 -> ands r3, #0  movs r3, #1 -> ands r3, #0/ add r3, # | 0.000147s | 0.000136s | 7.4% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Cmp r3,#0 thành tst r3,r3 | 0.004440s | 0.004367s | 1.6% |
| 3 | In ra giá trị các biến x,y,z | Tính giá trị chuỗi đầu vào và in ra giá trị các biến đã cho | Sử dụng biến thay thế | str r3, [r7, #16] ; đặt biến chung tính hàm strlen và lưu lại | 0.002176s | 0.001888s | 13.2% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế lệnh movs thành ands | 0.002560s | 0.002203s | 13.9% |
| 4 | Tính tổng n số bắt đầu từ m | Tính tổng n số bắt đầu từ m | Sử dụng biến thay thế | str r3, [r7, #8] ; dat bien chung co gia tri bang m+n | 0.001419s | 0.1395s | 1.6% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế các lệnh movs thành các lệnh ands và adds | 0.001496s | 0.001433s | 4.2% |
| 5 | Tính trung bình cộng của dãy số | Tính trung bình cộng của dãy số từ 1 đến 1000000 | Unroll loop | adds r3, r3, #4 ; tăng biến đếm lên 4 để giảm vòng lặp | 0.187196s | 0.169373s | 9.5% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands | 0.210628s | 0.176317s | 16.2% |
| 6 | In ra ký tự trong mảng | In ra ký tự trong mảng | Tối ưu kích thước bằng cờ Os | Sử dụng cờ Os để tối ưu kích thước | 0.002867s | 0.002867s | 0% |
| Tối ưu kích thước bằng cờ Os | Sử dụng cờ Os để tối ưu kích thước | 0.002395s | 0.002395s | 0% |
| 7 | Tính giai thừa | Tính giai thừa từ 1 tới n | Thay thế các lệnh phức tạp bằng các lệnh tương đương | bne .L3 ;  thay thế lệnh ble bằng bne | 0.001891s | 0.001781s | 5.8% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh cmp r3,#0 thành tst r3,r3 | 0.001789s | 0.001710s | 4.4% |
| 8 | Kiểm tra số nguyên tố | Kiểm tra xem số n đầu vào có phải số nguyên tố không | Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands và adds | 0.001768s | 0.001754s | 1% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands và cmp thành tst | 0.001751s | 0.001645s | 6% |
| 9 | Tính toán các biểu thức | Tính toán phép tính của 4 số cho trước . | Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands và adds | 0.000259s | 0.000185s | 28.5% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands và adds | 0.000212s | 0.000155s | 26.8% |
| 10 | Tính toán các biểu thức với các số nguyên cho trước | Tính tổng từ 1 đến n, tổng giai thừa 1! đến n! và một số biểu thức liên quan. | Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands và adds | 0.002423s | 0.001937s | 20% |
| Thay thế các lệnh phức tạp bằng các lệnh tương đương | Thay thế câu lệnh movs thành ands và adds | 0.002241s | 0.001781s | 20.5% |

# KẾT LUẬN

Sau thời gian tìm tòi, nghiên cứu và thực hiện, em đã hoàn thành đề tài “TỐI ƯU HÓA VÒNG LẶP TRONG ASSEMBLY”.

Cùng với sự phát triển như vũ bão của công nghệ thông tin, các thế hệ vi điều khiển ngày càng phát triển không ngừng nhằm đáp ứng các yêu cầu điều khiển, xử lý dữ liệu ngày càng lớn. Việc tìm hiểu, nghiên cứu về tối uu hóa phần mềm nhúng là một trong những nhiệm vụ cần thiết để có đủ kiến thức làm việc với các hệ thống nhúng. Qua bài báo cáo môn Tối ưu hóa phần mềm nhúng với đề tài: “Tối ưu hóa vòng lặp trong assembly”, em đã có cơ hội tìm hiểu và nghiên cứu các phương pháp để cải thiện một phần mềm nhúng về cả hiệu năng và kích thước.

Kết thúc bài báo cáo, em xin cảm ơn thầy Phạm Văn Hưởng đã giúp đỡ chỉ dẫn em có thể học hỏi thêm nhiều kiến thức mới cũng như đã có thể áp dụng chúng vào báo cáo một cách thành công.

# TÀI LIỆU THAM KHẢO:

* ***Tài liệu tiếng Việt:***

[1] TS. Phạm Văn Hưởng, *”Bài giảngTối ưu phần mềm nhúng”, Học viện Kỹ thuật Mật mã*

* ***Tài liệu tiếng Anh:***

[1]Agner Fog., “*Optimizing subroutines in assembly language”,*Technical University of Denmark, *2020, page 59-66*