Bộ nhớ và hiệu suất chương trình

Trong 1 lần đi sâu vào hiệu suất chương trình, Eric Brumer đã giải thích tại sao bộ nhớ thường là thành phần quan trọng nhất.

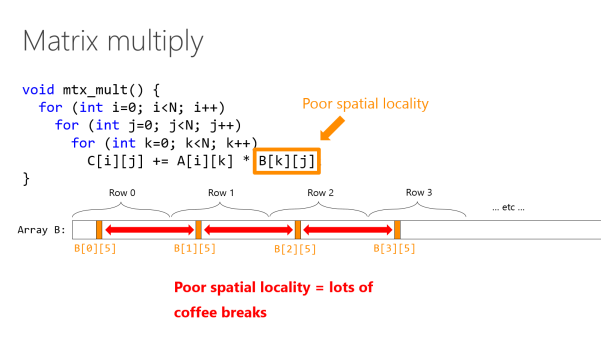
# Tổng quan về bộ nhớ đệm (Cache)

Với mỗi nhân bộ vi xử lý Intel Sandy Bridge có 6 đường ống (pipeline), 64 kB bộ nhớ đệm L1 và 256 kB bộ nhớ đệm L2. Bộ nhớ đệm L3 chia sẻ giữa 4 nhân có dung lượng từ 8 đến 30 MB tùy theo từng loại.

Khi đọc 1 số thực 32 bit từ bộ nhớ L1 cần 4 chu kỳ (cycle), từ bộ nhớ L2 là 12 chu kỳ và từ bộ nhớ L3 là 26 chu kỳ. Nếu đọc từ bộ nhớ thường thì cần thời gian nhiều hơn nữa.

# Truy cập dữ liệu

Eric đưa ra 1 ví dụ về ảnh hưởng của thời gian truy cập dữ liệu đến hiệu suất:



Hình 1. Ảnh hưởng của tốc độ truy cập

Chỉ bằng việc đổi chỗ j và k trong ví dụ trên, tốc độ chạy có thể tăng 10 lần đối với máy có cấu hình cao, 18 lần với máy có cấu hình thấp.

# Hiệu quả với vi xử lý đa nhân

Giả sử chúng ta có 2 mảng, mỗi mảng có 4 triệu phần tử số thực. Tạo mảng thứ 3 bằng cách cộng lần lượt từng phần tử từ 2 mảng trên (c[i] = a[i] + b[i]; i = 0..3,999,999). Theo lý thuyết, với bộ vi xử lý 5 nhân, tốc độ sẽ tăng lên gấp 5 lần, nhưng Eric thấy rằng tốc độ chỉ tăng 1.6 lần (32% lý thuyết). Tăng lên 10 nhân, tốc độ chỉ tăng thêm 2.4 lần (24% lý thuyết).

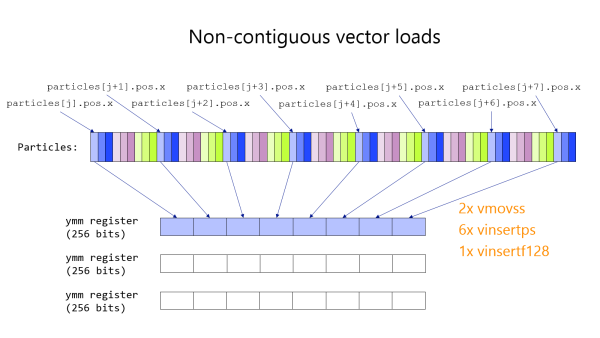
Nhìn vào kết quả, ông ấy phát hiện ra rằng toàn bộ thời gian được dùng để đọc dữ liệu từ bộ nhớ. Ba mảng có dung lượng tổng cộng là 48MB, lớn hơn 30MB dung lượng của bộ nhớ đệm L3 trong bộ vi xử lý 10 nhân trên. Bằng việc dùng 2 CPU với tổng cộng 20 nhân và 60MB bộ nhớ đệm L3, tốc độ thực hiện tăng lên 18 lần (90% lý thuyết).

Qua đó ta thấy được tầm quan trọng của bộ nhớ đệm. Hầu hết thời gian các chương trình chạy sẽ nhanh hơn khi chạy trên 1 CPU do các luồng xử lý có thể dễ dàng chia sẻ qua bộ nhớ đệm.

# Vector code

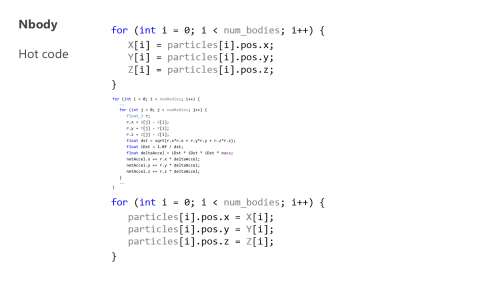
Trong ví dụ tiếp theo, Eric có 4096 hạt vật lý tương tác với nhau. Sử dụng 1 vòng lặp bình thường, ông có được tốc độ 8 khung hình mỗi giây. Nếu sử dụng vector trong C++ và tập lệnh SSE2 128-bit, số lần lặp giảm 1 nửa và hiệu suất tăng lên 18 khung hình mỗi giây. Với tập lệnh AVK 256-bit, số lần lặp giảm 1 nửa một lần nữa nhưng hiệu suất chỉ tăng lên 23 khung hình mỗi giây.

Vấn đề ở đây là dữ liệu đã được cấu trúc như thế nào. Ta thấy mỗi tọa độ X, Y, Z được lưu trữ tách rời nhau. Để tải 1 thanh ghi YMM 256-bit chứa tọa độ X của 8 hạt cần 9 lệnh cần thực hiện. Tương tự với toạn độ Y và Z cần 18 lệnh.



Hình 2. Vector code

Eric đã sao chép các tọa độ vào 3 mảng tạm thời, thay đổi vòng lặp chính sử dụng các mảng mới tạo, sao chép tọa độ mới trở lại các đối tượng. Với cùng việc sử dụng tập lệnh AVK 256-bit, tốc độ đã tăng lên 42 khung hình mỗi giây.



Hình 3. Sử dụng các mảng tạm

# Sắp xếp

Vấn đề tiếp theo là sự sắp xếp trong bộ nhớ. Mỗi line bộ nhớ đệm L1 có độ rộng 64 bytes. Nếu kích cỡ mỗi kiểu dữ liệu là ước số của độ rộng mỗi line (ví dụ: 32, 16, 8, …) thì ta có thể xếp chúng vào vừa một line mà không gây lãng phí bộ nhớ. Nhưng điều đó chỉ xảy ra nếu các phần tử được xếp theo mỗi 64 bytes.

Quay lại ví dụ trước, Eric đã cố truy cập mà không sắp xếp bằng cách bỏ qua phần tử đầu tiên của mảng. Kết quả là hiệu suất giảm 8% so với việc bắt đầu thực hiện vòng lặp với j bằng 0.

Với C tiêu chuẩn, malloc chỉ trả về dữ liệu sắp xếp mỗi 8 bytes. Để nhận được dữ liệu sắp xếp 32 hoặc 64 bytes ta có thể sử dụng \_aligned\_malloc. Hiện nay các trình biên dịch tự động gọi \_aligned\_malloc khi dữ liệu được sử dụng trong vòng lặp, nhưng việc đó có thể làm gia tăng sự phân mảnh dữ liệu.