**UTILIZAÇÃO DAS EXTENSÕES MULTIMÍDIA DOS PROCESSADORES INTEL® PARA REDUÇÃO DO NÚMERO DE CICLOS PARA A EXECUÇÃO DE PROGRAMAS**

**HOLANDA,** Adriano de Jesus[[1]](#footnote-2),[[2]](#footnote-3)

**RUIZ,** Evandro Eduardo Seron2

**CARNEIRO,** Antônio Adilton[[3]](#footnote-4)

**RESUMO:** A utilização das extensões multimídias com registradores que realizam a mesma operação sobre vários dados ao mesmo tempo (SIMD) dos atuais processadores podem reduzir o tempo de execução de programas que lidam com operações aritméticas sobre grande quantidade de dados. O objetivo deste trabalho foi quantificar o número de ciclos utilizados para o cálculo da correlação cruzada em duas dimensões para várias séries geradas e de diferentes tamanhos, usando a linguagem de programação C e as extensões para cálculo multimídia em Assembly para a codificação das instruções, compilação e execução. A comparação entre os resultados, usando o mesmo algoritmo e conjunto de dados, demonstrou que o programa em Assembly usando a extensão SIMD utilizou 38,37% menos ciclos de processador que a mesma implementação escrita em C.

**Palavras-chave:** Assembly. Processador. SIMD. Correlação cruzada.

**USE OF MULTIMEDIA EXTENSIONS OF INTEL® PROCESSORS TO DECREASE THE NUMBER OD CYCLES OF A PROGRAM EXECUTION**

**SUMMARY:** The use of multimedia extension with registers that perform the same operation in multiple data (SIMD) of the current processors should decrease the execution time of programs used to perform the same operation in a large quantity of data. The aim of the work was quantify the number of cycles needed to perform two-dimensional cross correlation calculation on a number of generated series with different number of elements, using a program compiled using C programming language and another one compiled using Intel® Assembly x86-64 language and the multimedia extensions. The Assembly program performed the same operation of the C program using in average 38.37 % less processor cycles.

**Keywords:** Assembly. Processor. SIMD. Cross correlation.

1. INTRODUÇÃO

Mesmo com a disseminação dos processadores com arquitetura de 64 bits substituindo os de 32 bits, as demandas por maior capacidade de armazenamento temporário nos registradores e melhor eficiência na operação sobre os dados armazenados vêm pressionando os fabricantes de processadores a adotarem soluções que se desviam da arquitetura normalmente empregada para construção dos processadores.

A principal modificação é a inclusão de registradores de 128 e 256 bits para a realização de vários cálculos de uma única vez, aumentando a performance dos programas, que utilizam estes recursos, causado pela redução no número de ciclos durante o processamento.

Estes recursos que começaram a ser manufaturados pela Intel®permitem que múltiplos dados sejam manipulados por uma única instrução (SIMD – *Single Instruction Multiple Data*), e fazem parte da tecnologia de extensão multimídia (MMX – *Multimedia Extension*) introduzida a partir do Pentium II, para realizar operações complexas sobre números inteiros.

Uma extensão da tecnologia SIMD denominada SSE (*Streaming* SIMD *Extension*) possui registradores de 128 bits para o armazenamento de múltiplos dados. A Figura 1 mostra como dados do tipo inteiro de diferentes tamanhos podem ser armazenados em um registrador SSE (BLUM, 2005).

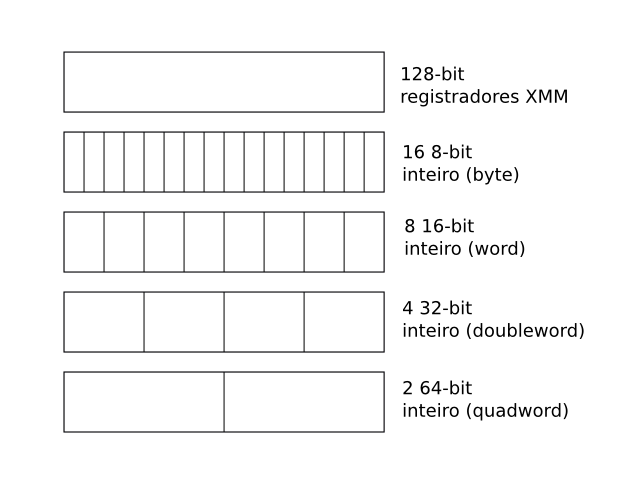


Figura 1 – Possibilidade de divisões dos registradores XMM de 128-bit para armazenamento de números inteiros.

(Fonte: Blum, 2005)

Utilizando os registradores de 128-bit para armazenamento de 4 números inteiros de 32 bits, que poderiam representar 4 pixels de um mapa de bits de imagens RGB (Red x Green x Blue), a soma de 4 números dois a dois pode ser realizada em 1 ciclo do processador, enquanto utilizando o método tradicional seriam necessárias dois ciclos do processador para a execução da operação. A Figura 2 ilustra como a operação em dados múltiplos é utilizada para redução do número total de ciclos para o processamento.

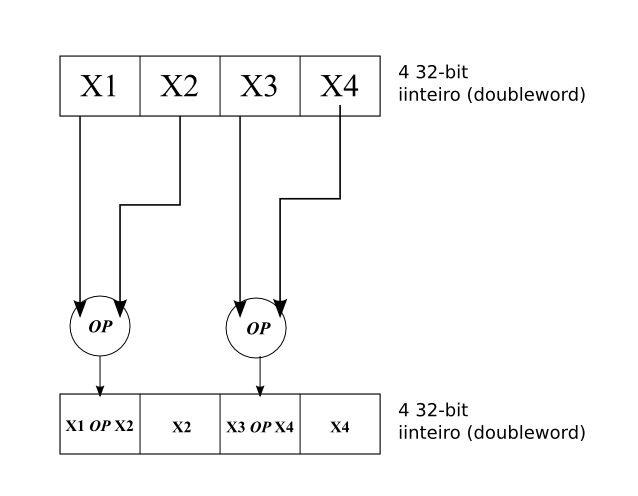


Figura 4 – Operação realizada em múltiplos dados sobre o mesmo registrador.

A Figura 4 ilustra a execução de uma mesma operação em vários dados, onde uma operação é realizada sobre o primeiro e segundo segmentos e o resultado armazenado no primeiro , enquanto a mesma operação é realizada sobre o terceiro e quarto segmentos com o armazenamento ocorrendo no terceiro segmento. Neste modo de execução, o segundo e quarto segmentos permanecem inalterados.

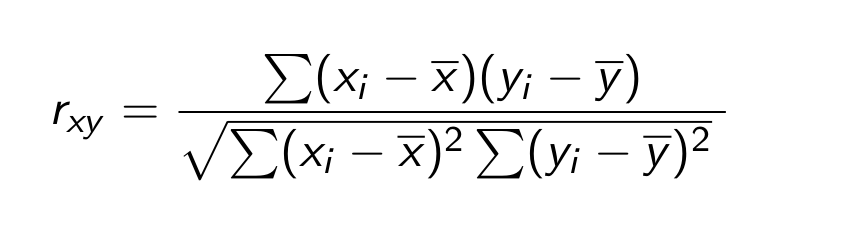
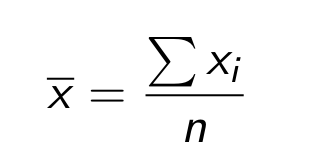
A utilização destes recursos para a execução de operações aritméticas podem reduzir o número de ciclos necessários para completar um tarefa, reduzindo o tempo de execução do programa.

1. OBJETIVO

O objetivo deste trabalho é quantificar a diferença no tempo de execução de instruções que utilizam a compilação na linguagem C sem o uso das extensões SSE e com uso da extensão SSE com registradores de 128 bits e divididos em inteiros de 32 bits.

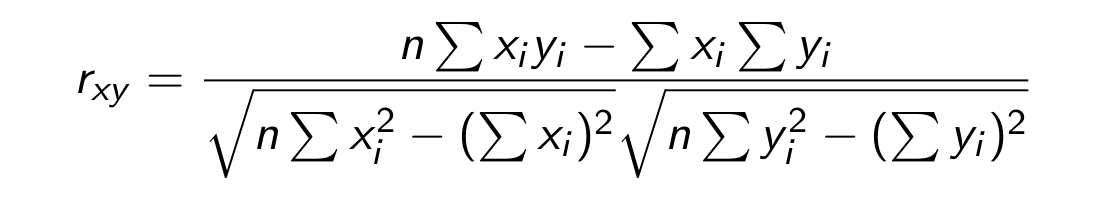
1. MATERIAL E MÉTODOS
   1. Codificação

O algoritmo escolhido para os testes é utilizado para o cálculo de correlação cruzada entre duas séries de valores igual tamanho, sua fórmula original é mostrada na Equação 1:

1. (1)
2. onde a média dos valores é calculada por
3. (2)

Porém, a Equação 1 tem complexidade *O(n2),* onde *n* é o número de elementos na série, pois há a necessidade de realizar duas passagens pelos dados, uma para calcular a média das séries *x* e *y*, e outra para calcular o valor da correlação cruzada *rxy* entre as séries.

A Equação 1 pode ser decomposta de modo a necessitar somente de uma passagem pelos dados, possuindo complexidade *O(n)* (Equação 3):



(3)

A correlação cruzada é comumente utilizada em processamento de sinais e imagens para analisar as similaridades entre as séries de dados que podem ser valores de pixels ou intensidade do sinal em determinado tempo.

A implementação da Equação 3 utilizando a linguagem C é apresentada na Figura 5 e foi sugerida por Seyfard (2012), e modificada pelos autores para operar sobre números inteiros:

|  |
| --- |
| double **xcorr**(long x[], long y[], n)  {  register long i;  register long sum\_x=0, sum\_y=0,  sum\_xx=0, sum\_yy=0, sum\_xy=0;  for (i = 0; i < n ; i++) {  register long xval = \*(x+i);  register long yval = \*(y+i);  sum\_x += xval;  sum\_y += yval;  sum\_xx += xval \* xval;  sum\_yy += yval \* yval;  sum\_xy += xval \* yval;  }  return ((double)(n\*sum\_xy-sum\_x\*sum\_y))/  (sqrt((n\*sum\_xx-sum\_x\*sum\_x)\*  (n\*sum\_yy-sum\_y\*sum\_y)));  } |

Figura 5 – Código em linguagem C do cálculo da correlação cruzada entre duas séries de dados.

O código implementado em Assembly para processadores Intel® x86 de 64 bits é mostrado na Figura 5. O código também segue, em linhas gerais, o descrito por Seyfarth (2012), com algumas modificações. A primeira é utilizar somente números inteiros para as séries, pois normalmente os valores de pixels na imagem são do tipo inteiro. A segunda modificação é uma verificação no cálculo final da correlação para evitar divisões por zero, que ; poderiam causar erro fatal ao final da execução do programa.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  | segment | .data |  |
| 2 | one | dq | 1 |  |
| 3 |  | segment | .text |  |
| 4 |  | global | xcorr\_nat |  |
| 5 | xcorr\_nat: |  |  |  |
| 6 |  | xor | r8,r8 |  |
| 7 |  | move | rcx, rdx |  |
| 8 |  | subpd | xmm0, xmm0 | ; zera sum\_x |
| 9 |  | movdqa | xmm1, xmm0 | ; zera sum\_y |
| 10 |  | movdqa | xmm2, xmm0 | ; zera sum\_xx |
| 11 |  | movdqa | xmm3, xmm0 | ; zera sum\_yy |
| 12 |  | movdqa | xmm4, xmm0 | ; zera sum\_xy |
| 13 |  | movdqa | xmm8, xmm0 | ; zera n |
| 14 |  | movdqa | xmm9, xmm0 | ; zera variável temporária |
| 15 | .loop: |  |  | ;; Main loop |
| 16 |  | movdqa | xmm5, [rdi+r8] | ; atribui valor do próximo x da série |
| 17 |  | movdqa | xmm6, [rsi+r8] | ; atribui valor do próximo y da série |
| 18 |  | paddq | xmm0, xmm5 | ; sum\_x |
| 19 |  | paddq | xmm1, xmm6 | ; sum\_y |
| 20 |  | movdqa | xmm7, xmm5 | ; atribui x a um registrador temporário |
| 21 |  | pmuldq | xmm7, xmm6 | ; x\*y |
| 22 |  | pmuldq | xmm5, xmm5 | ; x\*x |
| 23 |  | pmuldq | xmm6, xmm6 | ; y\*y |
| 24 |  | paddq | xmm2, xmm5 | ; sum\_xx |
| 25 |  | paddq | xmm3, xmm6 | ; sum\_yy |
| 26 |  | paddq | xmm4, xmm7 | ; sum\_xy |
| 27 |  | add | r8, 16 |  |
| 28 |  | sub | rcx, 2 |  |
| 29 |  | jnz | .loop |  |
| 30 |  | haddpd | xmm0, xmm0 | ; compacta sum\_x |
| 31 |  | cvtdq2pd | xmm0, xmm0 | ; converte sum\_x para ponto flutuante |
| 32 |  | haddpd | xmm1, xmm1 | ; compacta sum\_y |
| 33 |  | cvtdq2pd | xmm1, xmm1 | ; converte sum\_y para ponto flutuante |
| 34 |  | haddpd | xmm2, xmm2 | ; compacta sum\_xx |
| 35 |  | cvtdq2pd | xmm2, xmm2 | ; converte sum\_xx para ponto flutuante |
| 36 |  | haddpd | xmm3, xmm3 | ; compacta sum\_yy |
| 37 |  | cvtdq2pd | xmm3, xmm3 | ; converte sum\_yy para ponto flutuante |
| 38 |  | haddpd | xmm4, xmm4 | ; compacta sum\_xy |
| 39 |  | cvtdq2pd | xmm4, xmm4 | ; converte sum\_xy para ponto flutuante |
| 40 |  | cvtsi2sd | xmm8, rdx | ; armazena o valor de n em xmm8 |
| 41 |  | mulsd | xmm4, xmm8 | ; n\*sum\_xy |
| 42 |  | movsd | xmm9, xmm0 | ; tmp := sum\_x |
| 43 |  | mulsd | xmm9, xmm1 | ; sum\_x\*sum\_y |
| 44 |  | subsd | xmm4, xmm9 | ; numerador := n\*sum\_xy - sum\_x\*sum\_y |
| 45 |  | mulsd | xmm2, xmm8 | ; n\*sum\_xx |
| 46 |  | mulsd | xmm0, xmm0 | ; sum\_x\*sum\_x |
| 47 |  | subsd | xmm2, xmm0 | ; denominador0 := n\*sum\_xx - sum\_x\*sum\_x |
| 48 |  | sqrtsd | xmm2, xmm2 | ; denominador0 := sqrt(denominador0) |
| 49 |  | mulsd | xmm3, xmm8 | ; n\*sum\_yy |
| 50 |  | mulsd | xmm1, xmm1 | ; sum\_y\*sum\_y |
| 51 |  | subsd | xmm3, xmm1 | ; denominador1 := n\*sum\_yy - sum\_y\*sum\_y |
| 52 |  | sqrtsd | xmm3, xmm3 | ; denominador1 := sqrt(denominador1) |
| 53 |  | mulsd | xmm2, xmm3 | ; denominador := denominador0 \* denominador1 |
| 54 |  | movsd | xmm9, xmm2 | ; tmp := denominador |
| 55 |  | comisd | xmm4,xmm9 | ; denominador == numerador ? |
| 56 |  | jz | DEN\_EQ\_NUM | ; (denominador == numerador) := verdadeiro |
| 57 |  | divsd | xmm4, xmm2 | ; correlação := numerador/denominador |
| 58 |  | movsd | xmm0, xmm4 | ; retorna correlação |
| 59 |  | jz | EXIT |  |
| 60 | DEN\_EQ\_NUM: | movsd | xmm0, [one] | ; correlação := 1 |
| 61 |  | cvtdq2pd | xmm0, xmm0 | ; converte inteiro para ponto flutuante |
| 62 | EXIT: |  |  |  |

Figura 6 – Código em Assembly Intel® x86-64 para o cálculo da correlação cruzada entre duas séries de números inteiros.

Na Figura 6 o código em Assembly Intel® x86-64 é mostrado com o número da linha de cada instrução na primeira coluna, os rótulos na segunda coluna, as operações na terceira, os operandos na quarta e os comentários na quinta após o símbolo “;”.

Das linhas 15 à 29 estão as instruções para a realização do somatório para o cálculo da correlação cruzada conforme mostrado na Equação 3. As instruções paddq e pmuldq executam a soma e multiplicação, respectivamente, de quatro valores inteiros conforme mostrado na Figura 4. Das linhas 30 à 62, há instruções para a preparação dos resultados das somas, convertendo-os para ponto flutuante, para efetuar a divisão, e realizando cálculo final da correlação. A linha 56 garante que se o numerador e denominador da equação forem iguais a zero, ou seja a correlação seja igual a 1, a divisão pelo denominador igual a zero, não seja realizada retornando o valor 1 sem efetuar a divisão final.

Os códigos mostrados neste trabalho fazem parte um projeto de análise elastográfica de imagens de ultrassom desenvolvido pelo Departamento de Física da Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto/USP, e podem ser obtidos pelo site <https://github.com/ajholanda/nixus>.

* 1. Compilação e execução

O programa em C foi compilado usando o gcc (GNU *Compiler Collection,* <http://gcc.gnu.org/>) sem nenhuma *flag* adicional passada para o compilador. O código em Assembly, conforme mostrado na Figura 2, foi compilado usando o assembler Yasm (<http://yasm.tortall.net/>) com as seguintes *flags*: “-Worphan-labels -f elf64 -g dwarf2 -l corr.lst”. Os programas foram compilado e executados em um computador com processador Intel® CoreTM 2 Duo com 2,93 GHz e 3MB de memória cache, e memória DRAM com 3GB de capacidade.

O número de valores atribuídos para as séries foram gerados a partir dos índices do arranjo contendo os elementos com as seguintes atribuições:

x[i] = 1+i;

y[i] = 1+2\*i;

onde i é o índice do arranjo onde está sendo efetuado o *loop*.

Os cálculos de correlação entre as séries foram efetuados para os seguintes números de elementos:

<1.000,1.000,9.000>,

<10.000, 10.000, 90.000>,

<1.000.000, 1.000.000, 9.000.000>,

<10.000.000, 10.000.000, 90.000.000>,

onde o primeiro valor da tupla indica valor inicial para o *loop*, o segundo o passo e o terceiro o limite do *loop*.

1. RESULTADOS

A Figura 7 contêm a relação entre os número de pontos das séries x e y e o número de ciclos utilizados para calcular a correlação entre as séries utilizando o programa em C e A description...Assembly.

Figura 7 – Número de ciclos utilizados pelo processador em função de número de elementos das séries geradas para o cálculo da correlação cruzada.



Em média houve uma redução de 38,37% do número de ciclos utilizando o código em Assembly quando comparado com o código em C. Para um número menor de valores na série, a performance do código em Assembly é maior devido ao fato da menor utilização da memória cache. Para o número de elementos na série entre 10.000 e 100.000, a redução do número de ciclos usando o código em Assembly varia de 64,5% a 94,4. A partir de 100.000 o a redução dos valores de uso de ciclos do processador cai para menos de 40%.

1. DISCUSSÃO

Quando se tem por objetivo melhorar a performance dos programas que processam grande quantidade de dados, caso comum em processamento de imagens e sinais, o conhecimento do hardware utilizado para execução das tarefas pode contribuir para identificar pontos de otimização.

As linguagens de alto nível como o C fornecem um nível de abstração e portabilidade que contribuem para facilitar a codificação com maior facilidade de gerenciamento do código fonte. Porém, alguns recursos recentes do processadores modernos ainda não estão disponíveis nas linguagens de programação mais comuns. Por este motivo, a integração de código de alto nível com Assembly, usando esta última em pontos que exijam maior performance pode ser uma solução para reduzir o tempo de processamento total.

O processador Intel® CoreTM i7 possui registradores de 256 bits para a realização das operações em múltiplos dados, através de sua extensão AVX *(Advanced Vector Extensions),* possuindo operações específicas para dados multimídia.

Outros pontos de otimização mais ligados ao sistema operacional, como por exemplo melhor controle do sistema de entrada/saída (E/S), também cooperam para a redução do tempo total de processamento. Por exemplo, a manipulação do *buffer*, uso de operações de E/S não bloqueante e mapeamento do arquivo contendo os dados para memória principal podem reduzir o tempo de acesso aos dados, aumentando o fluxo de instruções no processador.

1. CONCLUSÃO

O uso da extensão SSE do processador Intel® para o cálculo da correlação cruzada entre duas séries de valores inteiros, de diferente tamanhos e com a geração do número inteiro baseada no índice do elemento, reduziu o número de ciclos utilizados no processador pelos programas em média 38,37%, número este que pode ser maior para computadores com alta capacidade de memória cache.

1. REFERÊNCIAS

BLUM, Richard. **Professional Assembly Language.**  Editora Wrox, 2005.

SEYFARTH, Ray. **Introduction to 64 Bit Intel Assembly:** Language Programming for Linux, 2012 (Edição Kindle).

1. Faculdade “Dr. Francisco Maeda” - FAFRAM, Ituverava/SP. [↑](#footnote-ref-2)
2. Departamento de Computação e Matemática, Faculdade de Filosofia Ciências e Letras de Ribeirão Preto (FFCLRP) /USP. [↑](#footnote-ref-3)
3. Departamento de Física, FFCLRP/USP. [↑](#footnote-ref-4)