



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

RELATÓRIO

Vetorização

Alexandre Henrique Soares Dias: 20180001100

José Tobias Souza dos Santos: 20180154969

Gabriel Medeiros Coelho: 20180009538

Natal-RN

2019

1. Introdução

1.1 Contextualização e motivação

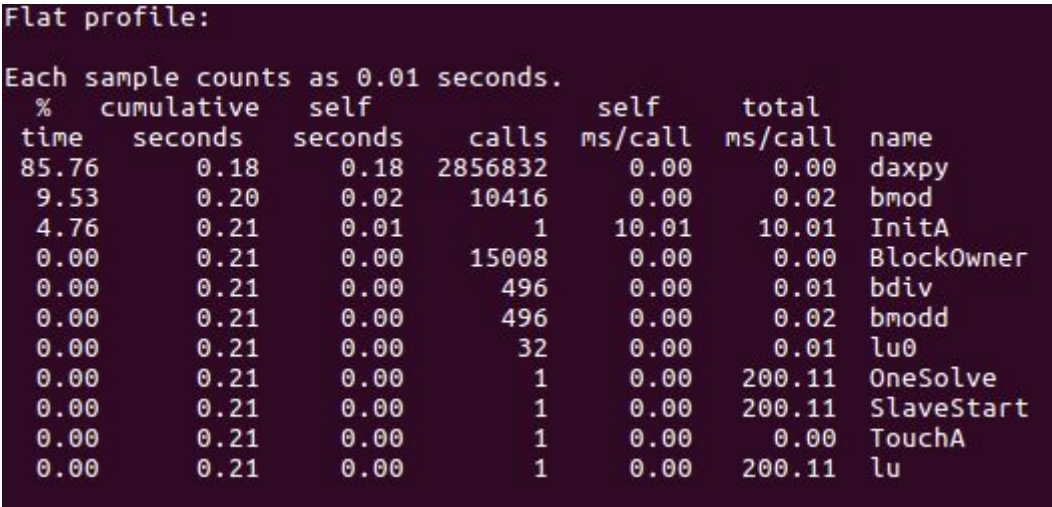
Atualmente, muitas CPUs utilizam-se de conjuntos operações vetoriais sobre seus registradores como um modo de otimizar a execução de programas e tarefas. A taxonomia de Flynn descreve uma classe de processadores chamada de SIMD, que é a terminologia atribuída a processadores que realizam uma única operação sobre múltiplos dados. Neste contexto, a partir de operações vetoriais, é possível otimizar a execução de algoritmos fazendo com que alguma(s) parte(s) dele sejam executadas sob a abordagem SIMD. Assim, o ato de realizar uma instrução sobre um conjunto de dados (*arrays*) ao invés de um único dado é chamado de **vetorização**.

Simplificando, vetorização é o processo de “reescrever” um *loop* de tal maneira que ao invés de processar apenas um elemento de um array M vezes, são processados 4 elementos simultaneamente M/4 vezes, por exemplo.

Buscando otimizar a execução da aplicação LU-CB da suíte *splash2*. Neste relatório, detalhamos os procedimentos adotados para vetorizar os *loops* contidos em regiões consideradas críticas do algoritmo, levando em consideração o tempo de execução despendido nestas regiões.

1.2 Perfilamento

Buscando caracterizar e identificar as regiões do código que tomam a maior parte do tempo de execução, em um momento anterior realizamos o perfilamento utilizando-se da ferramenta Gprof e obtivemos o seguinte resultado:



```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self           calls   self   total    name
time   seconds    seconds                ms/call  ms/call  name
85.76    0.18      0.18    2856832      0.00    0.00  daxpy
 9.53    0.20      0.02     10416      0.00    0.02  bmod
 4.76    0.21      0.01         1     10.01   10.01  InitA
 0.00    0.21      0.00     15008      0.00    0.00  BlockOwner
 0.00    0.21      0.00        496      0.00    0.01  bdiv
 0.00    0.21      0.00        496      0.00    0.02  bmodd
 0.00    0.21      0.00         32      0.00    0.01  lu0
 0.00    0.21      0.00          1      0.00   200.11  OneSolve
 0.00    0.21      0.00          1      0.00   200.11  SlaveStart
 0.00    0.21      0.00          1      0.00    0.00  TouchA
 0.00    0.21      0.00          1      0.00   200.11  lu
```

%	cumulative	self		self	total	name
time	seconds	seconds	calls	ms/call	ms/call	
85.76	0.18	0.18	2856832	0.00	0.00	daxpy
9.53	0.20	0.02	10416	0.00	0.02	bmod
4.76	0.21	0.01	1	10.01	10.01	InitA
0.00	0.21	0.00	15008	0.00	0.00	BlockOwner
0.00	0.21	0.00	496	0.00	0.01	bdiv
0.00	0.21	0.00	496	0.00	0.02	bmodd
0.00	0.21	0.00	32	0.00	0.01	lu0
0.00	0.21	0.00	1	0.00	200.11	OneSolve
0.00	0.21	0.00	1	0.00	200.11	SlaveStart
0.00	0.21	0.00	1	0.00	0.00	TouchA
0.00	0.21	0.00	1	0.00	200.11	lu

A imagem obtida como *output* indica que as funções *daxpy* e *bmod* são as que demandam mais tempo de processamento.

Por essa razão, as regiões de código dessas duas funções foram consideradas críticas, e tornaram-se alvo de uma análise minuciosa a fim de descobrirmos possíveis técnicas que poderiam ser adotadas para acelerar suas execuções.

2. Vetorização

Visando o aumento de desempenho na execução das regiões críticas, no comando de compilação do algoritmo foram adicionadas *flags* para indicar ao compilador GCC que desejamos que o mesmo realize a auto-vetorização dos *loops*. O comando de compilação passado no *shell* foi o seguinte:

```
$ gcc -lpthread -g -lm -o lu.o lu.c -O3 -fvectorize -fopt-info-vec-all -funsafe-math-optimizations 2> output.txt
```

As *flags* mais relevantes no tocante a otimização do algoritmo são:

1. **-O3**: Habilitar um conjunto amplo de diretivas de otimização
2. **-fvectorize**: Indicar ao compilador que ele deve tentar vetorizar *loops*
3. **-fopt-info-vec-all**: Emitir relatório de vetorização
4. **-funsafe-math-optimizations**: Otimização para operações aritméticas em ponto flutuante

2.1 *daxpy*

O corpo da função *daxpy* é mostrado ao lado.

```
748 void daxpy(a, b, n, alpha)
749
750 double *a;
751 double *b;
752 double alpha;
753 int n;
754
755 {
756     int i;
757     for (i=0; i<n; i++) {
758         a[i] += alpha*b[i];
759     }
760 }
761 }
```

Esta função contém apenas um *loop*. O compilador conseguiu vetorizar este laço com sucesso e no arquivo de saída obtivemos o resultado:

lu.c:758:3 = LOOP VECTORIZED

2.2 *bmod*

Por outro lado, para a função *bmod* os resultados obtidos não foram tão diretos. Abaixo, o corpo da função pode ser visualizado:

```
721 void bmod(a, b, c, dimi, dimj, dimk, stridea, strideb, stridec)
722
723 double *a;
724 double *b;
725 double *c;
726 int dimi;
727 int dimj;
728 int dimk;
729 int stridea;
730 int strideb;
731 int stridec;
732
733 {
734     int i;
735     int j;
736     int k;
737     double alpha;
738
739     for (k=0; k<dimk; k++) {
740         for (j=0; j<dimj; j++) {
741             alpha = -b[k+j*strideb];
742             daxpy(&c[j*stridec], &a[k*stridea], dimi, alpha);
743         }
744     }
745 }
```

Desta vez, temos 2 laços aninhados, e o resultado obtido para a tentativa de vetorização desses laços foi:

lu.c:739:3 = not vectorized:

multiple nested loops

lu.c:740:5 = not vectorized:

control flow in loop

Isto é, nenhum dos *loops* foi vetorizado. Identificamos que a chamada da função *daxpy* dentro do laço interno é a possível causa do laço não poder ser vetorizado. Daí vem a expressão *control flow in loop* provida na saída do relatório de vetorização. E ainda, como a função *daxpy* invoca um outro laço, isso faz com que o compilador não consiga vetorizar o laço mais externo, emitindo sendo esta a possível causa do erro *multiple nested loops*.

A partir desta saída, apresentamos possíveis abordagens utilizando diretivas OpenMP SIMD para contornar o problema da não-vetorização dos laços.

1. **#pragma omp declare simd** (antes da declaração função): Esta diretiva transforma as entradas e saídas da função em variáveis vetoriais.
2. **#pragma omp simd** (antes do *loop*): Habilita as otimizações do OpenMP para tornar o laço mais “amigável” e, possivelmente, vetorizável.

3. Conclusões

Finalmente, uma vez que conseguimos vetorizar o *loop* da função *daxpy*, acreditamos que a vetorização nos trará um aumento considerável no desempenho do algoritmo levando em conta que esta função toma cerca de 85% do tempo de execução total do mesmo. Expresso de outra forma, possivelmente conseguimos uma melhoria considerável da execução através de um paralelismo em nível de dados, não apenas no nível de instrução.

Em contrapartida, *a priori*, não foi possível vetorizar os laços da função *bmod*. No entanto, as estratégias de otimização apresentadas serão empregadas em um momento futuro a fim de alcançarmos este objetivo.

Referências

WOO, Steven Cameron et al. **The SPLASH-2 programs: Characterization and methodological considerations.**

ACM SIGARCH computer architecture news, v. 23, n. 2, p. 24-36, 1995.

FLYNN, Michael J. Some computer organizations and their effectiveness. **IEEE transactions on computers**, v. 100, n. 9, p. 948-960, 1972.