Universidade Federal de Pelotas

Disciplina de Conceitos de Linguagens de Programação

Otimização de laços em C

Grupo: Geovana Silveira
Luiza Cruz
Vitor Gosmão

Pelotas, 22 de junho de 2018

Sumário

1. Introdução

Esse relatório possui como objetivo apresentar as otimizações de laços utilizadas no algoritmo base e os resultados obtidos através destas otimizações. A otimização de laços é o processo de aumento da velocidade de execução e redução das sobrecargas associadas aos laços. Efetuando um papel importante na melhoria do desempenho da memória cache e no uso efetivo da capacidade de processamento paralelo. Para o realizamento das otimizações no código nos laços serão utilizadas técnicas implementada pelo próprio compilador e estratégias de programação.

2. Estudo de Caso

Será utilizado como algoritmo base do problema de otimização de laços o algoritmo de ordenação de vetores chamado Selection Sort. Este algoritmo funciona da seguinte maneira: Primeiro ele acha o menor elemento do vetor e realiza uma troca de posição com o elemento da primeiro posição, até então atribuído como o elemento de menor valor do vetor de entrada. Depois passa o segundo menor elemento para a segunda posição, e assim sucessivamente até que todo vetor esteja ordenado. Este algoritmo é composto por dois laços, um laço externo e outro interno. O laço externo serve para controlar o índice inicial, já o laço interno serve para percorre o vetor. Na primeira iteração do laço externo o índice começa no 0 e a cada iteração ele soma uma unidade até o fim do vetor, já o laço mais interno percorre o vetor começando do índice externo + 1 até o final do vetor.

3. Implementação Trivial

3.1. Código em C

A implementação trivial do algoritmo de ordenação Selection Sort na linguagem C é como mostrada abaixo:

selectionSort.c

```
void selectionSort(int vet[], int tam){
   int i, j, aux, menor;
   for (i = 0; i < (tam-1); i++){
       menor = i;      /* Atribui como menor o primeiro número não ordenado ainda */
   for (j = (i+1); j < tam; j++){
         if(vet[j] < vet[menor]){ //busca pelo menor elemento através do índice
            menor = j;  //salva o novo índice como menor
```

```
        }
      }
      /* troca e coloca o menor elemento para frente */
      if(vet[i] != vet[menor]){
        aux = vet[i];
        vet[i] = vet[menor];
        vet[menor] = aux;
      }
   }
}
```

4. Otimização

4.1. Vetorização

Visando uma melhora de performance do tempo de execução do algoritmo Selection Sort, foi aplicada vetorização nos laços do algoritmo através das diretivas SIMD da API OpenMP. Essas diretivas utilizadas em loops permitem que o compilador execute os laços com vetorização.

4.1.1. Pragma OMP for SIMD

Para a otimização dos laços, foi utiliza a diretiva omp for simd, que especifica que um loop pode ser executado concorrentemente usando instruções simd, e que essas interações também serão executadas em paralelo pela thread do time.

selectionSimd.c

```
void selectionSort (int vet[],int tam){
  int i, j, menor, aux;
  #pragma omp for simd
  for(i = 0; i < (tam-1); ++i) {
    menor = i;  // o menor é o primeiro número não ordenado ainda
  for(j = (i+1); j < tam; ++j){
   if(vet[j] < vet[menor]) // busca pelo menor elemento através do índice
    menor = j;  // salva o novo índice como menor
  }
  /* troca e coloca o menor elemento para frente */
   if(vet[i] != vet[menor]){
     aux = vet[i];
     vet[i] = vet[menor];
     vet[menor] = aux;
```

```
    }
  }
}
```

4.1.2. Pragma SIMD Reduction

Foi utilizado a diretiva simd no qual a cláusula reduction realiza uma redução nas variáveis de dados de acordo com o identificador de redução. No caso, a redução é feita através da criação de uma cópia privada das variáveis com o menor valor de cada execução que está sendo realizada em paralelo e atualiza o valor das variáveis originais ao final do laço.

selectionReduction.c

```
/* Cria uma cópia privada para cada execução para armazenar o menor valor de cada vetor
em execução */
#pragma omp declare reduction(min : struct Compara : omp_out = omp_in.valor >
omp_out.valor ? omp_out : omp_in)

void selectionsort(int *vet, int tam){
    int i,j,aux;
    struct Compara menor;
     /* diretiva omp simd aplicada ao laço indica que múltiplas iterações do loop podem
ser executadas concorrentemente usando instruções SIMD */
    #pragma omp simd reduction(min:menor)  //vetorização com redução
   for (i = 0; i < (tam - 1); ++i){
      menor.valor = vet[i];
      menor.indice = i;  //armazena índice do menor elemento
   for (j = i + 1; j < tam; ++j){
      if (vet[j] < menor.valor){ // busca pelo elemento de menor valor
          menor.valor = vet[j];
          menor.indice = j; // salva o novo índice como menor
       }
   }
      /* troca e coloca o menor elemento para frente */
      aux = vet[i];
      vet[i] = menor.valor;
      vet[menor.indice] = aux;
   }
}
```

4.2. Flag na Compilação

Como vetorização é desabilitada por default pelo gcc, foi utilizada a flag -O2 durante a compilação, essa flag além de habilitar a vetorização também é responsável pela otimização dos laços de repetição do programa. O compilador realiza quase todas as otimizações suportadas que não envolvam uma troca de espaço-velocidade. Utilização da flag aumenta o tempo de compilação e a performance do código gerado.

5. Análise de Desempenho

Para a análise comparativa dos tempos de execuções do algoritmo base e das otimizações realizadas foram feitos testes utilizando vetores de tamanhos diferentes como mostrada na Figura 1, sendo este vetor formado de valores pseudo-aleatórios através da função rand. A escolha de tamanho de vetores grandes para os testes foi devido ao algoritmo apresentar lentidão para estes casos. Para o cálculo do tempo de execução foi usada a função clock da biblioteca time.h presente na linguagem c. Os dados relacionados ao consumo de memória utilizada foram coletados utilizando a ferramenta Valgrind.

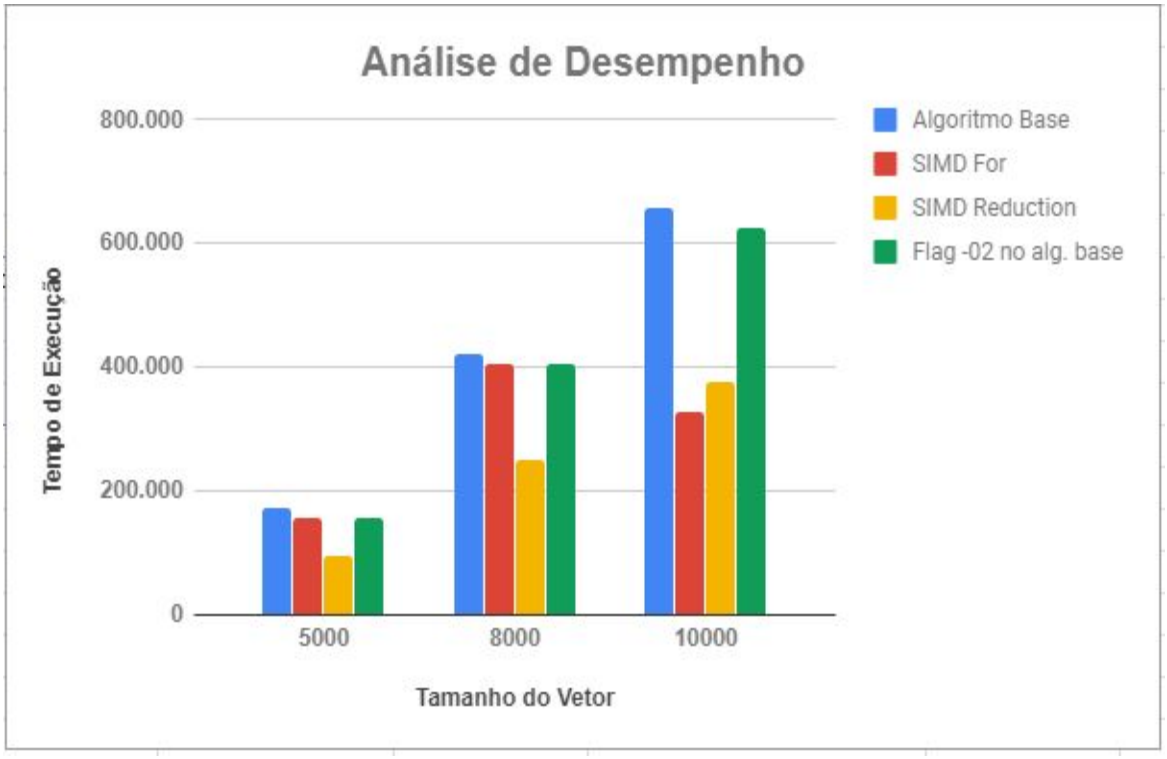

Figura 1: Comparativo do tempo de execução (em segundos) do algoritmo base em relação a otimizações realizadas.

Consumo de memória: Em sua maioria não foi apresentada grandes alterações no consumo de memória na execução das otimizações realizadas em relação ao algoritmo base, acredita-se que isso tenha acontecido devido ao fato do algoritmo do Selection Sort não possuir um melhor caso, passando sempre pelos dois laços do algoritmo original independente da formação inicial dos elementos no vetor. Tendo uma complexidade O(n^2) para todos os casos. Como se pode observar através das tabela 1, a otimização utilizando a diretiva omp simd for foi a que consumiu mais memória, isso se deve a sua implementação utilizar um tipo referenciando uma struct, o que o difere do algoritmo base.

Tamanho do Vetor	Algoritmo Base	Simd For	Sind Reduction	Flag -O2 no algoritmo base
5000	28.192	61.016	64.192	28.192
8000	40.192	73.016	40.192	40.192
10000	48.192	81,016	48.192	48.192

Tabela 1: Quantidade (em bytes) de memória heap utilizada na execução do algoritmo base e das otimizações realizadas.

Tempo de execução: Como se pode observar na tabela 2, as otimizações realizadas no algoritmo base do Selection Sort resultaram em tempos de execução menores do que o tempo do algoritmo original, com a otimização utilizando a diretiva #pragma omp simd reduction da API OpenMP apresentando o menor tempo entre as três otimizações. Mostrando que a utilização do paralelismo através das instruções SIMD para realizamento das otimizações sobre os laços presentes no algoritmo de ordenação Selection Sort além de reduzir a quantidade de ciclos por instrução, garantindo assim uma execução mais rápida do algoritmo também possibilita uma maior capacidade de exploramento dos recursos disponíveis pela arquitetura utilizada.

Tamanho do Vetor	Algoritmo Base	Simd For	Sind Reduction	Flag -O2 no algoritmo base
5000	171.875	156.250	93.750	156.250
8000	421.875	406.250	250.000	406.250
10000	656.250	328.125	375.000	625.000

Tabela 2: Tempo de execução (em segundos) do algoritmo base e de suas otimizações para os determinados tamanhos de vetores utilizados para a realização dos testes.

Referências Bibliográficas

*JEFFERS**, J.; REINDER, J.; SODAN, A.** ***Intel Xeon Phi Processor High Performance programming: Knights*** **Landing Edition**. 2nd Edition.

SEDGEWICK, Robert. ***Algorithms in C****, 3rd. edition, vol. 1, Addison Wesley Longman, 1998. ISBN 0201314525.*

IBM Knowledge Center. #pragma omp simd*. Disponivel em : <https://www.ibm.com/support/knowledgecenter/en/SSXVZZ_16.1.0/com.ibm.xlcpp161.lelinux.doc/compiler_ref/prag_omp_simd.html>. Acesso em: 15 de junho de 2018.*

A Guide to Vectorization with Intel® C++ Compilers. *Disponivel em: <https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf >. Acesso em: 16 de junho de 2018.*

Using the GNU Compiler Collection(GCC): Optimize Options*. Disponivel em: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html >. Acesso em: 20 de junho de 2018.*