Análise de Vetorização do Algoritmo *Streamcluster*

Angelo Leite Medeiros Goes
20200000545
Jhonat Heberson Avelino de Souza
20200000680
Angelo Marcelino Cordeiro
20190152879
Maurício Thiago Ferreira de Lima
20180155222

I. Estrutura de regiões paralelas e laços da aplicação

O fluxo paralelo do algoritmo *streamcluster* [1] se resume na divisão de dados de entrada na forma de pontos, e no cálculo constante de distâncias, medianas, custo de criação de novos centroides, e demais gastos relacionados ao custo-benefício de prosseguimento do algoritmo, ou seja, durante o processo de decisão que dirá se o novo centroide é melhor que os anteriores. Também, o fluxo paralelo se encaixa na utilização de barreiras de *threads* [2] que sincronizem as operações realizadas.

Sabendo disso, é possível observar duas regiões distintas na estrutura do código: uma de inicialização (até a função `localSearch()`) e uma paralela, onde se concentram os laços da aplicação, destacada pelo pontilhado azul na Figura 1.

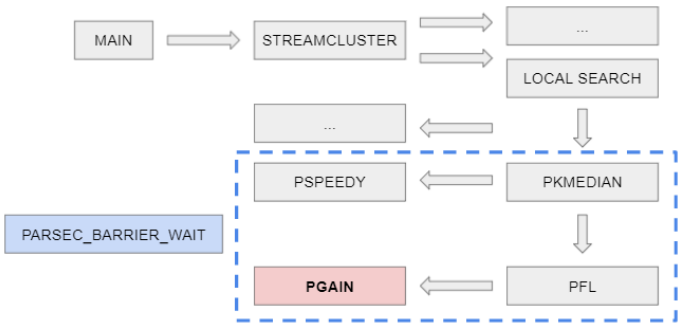

Fig. 1. Abstração estrutural de fluxo

O programa parte da função `streamcluster()` que, dentre outras funções de inicialização e tratamento de dados, chama `localSearch()`, ou busca local, que é responsável por inicializar as *threads*, atribuir certas variáveis iniciais, e prosseguir com as demais funções aritméticas, dentro da região paralela.

Haverá permanência nesses laços até que se encontre um número ótimo de centroides (dentro do intervalo [`kmin`, `kmax`] passado como parâmetro inicial) e localizações aceitáveis para os mesmos.

Os laços se distribuem majoritariamente no processo de cálculo de centroides, dentro das funções `pspeedy()`, `pkmedian()`, `pFL()` e `pgain()`, e durante a criação e manipulação de threads, nas funções `streamcluster()` e `localSearch()`.

II. Detalhes das flags usadas para auto-vetorizar os laços

Na documentação do compilador `gcc` [3] foram encontrados vários tipos de *flags* que possuem o intuito de otimizar o código e que mostram as informações do resultado dessa otimização, um exemplo dessas *flags* pode ser demonstrado a seguir:

- `-O3 -fopt-info-vec-all` - Mostra todas as informações
- `-O3 -fopt-info-vec-missed` - Mostra somente as informações não-otimizadas

- `-O3 -fopt-info-vec-optimized` - Mostra somente as informações otimizadas

Onde a opção `-O3` é responsável por habilitar a vetorização. Ela faz parte dos níveis de otimização, que podem variar em seus objetivos e características. Como pode-se ver na Figura 2, outro compilador, *armclang* [4], caracteriza seus níveis de otimização dentre vários escopos, entre eles: melhor performance e menor tamanho do código, entre outros. No caso do compilador `gcc`, especificamente a opção `-O3` habilita *flags* que procuram vetorizar o código: `-ftree-loop-vectorize` e `-ftree-slp-vectorize`.

Table 3-8 Optimization goals

Optimization goal	Useful optimization levels
Smaller code size	`-Oz`, `-Omin`
Faster performance	`-O2`, `-O3`, `-Ofast`, `-Omax`
Good debug experience without code bloat	`-O1`
Better correlation between source code and generated code	`-O0` (no optimization)
Faster compile and build time	`-O0` (no optimization)
Balanced code size reduction and fast performance	`-Os`

Fig. 2. Tabela de Objetivos de Otimização [4]

Foi utilizada a combinação de *flags* `-O3 -fopt-info-vec-all`, visando encontrar quais partes do código geram problemas ao tentar realizar o processo de auto-vetorização. Os resultados caíram nas seguintes categorias: Fluxo de controle dentro do *loop*; Declaração não suportada; Custo não compensa; Variáveis do tipo volátil; Número de iterações não pôde ser computado; Presença de *"memory clobbers"*.

Ao final de sua execução e análise, foram totalizados 47 problemas com 22 resultantes de laços não-vetorizados e 25 problemas em partes de código dependentes de barreiras de *threads*. Dentre todos os problemas, cerca de 80% estão nas funções `pgain()` e `pspeedy()`.

III. Lista de laços não vetorizados com estudo de estratégias para vetorizar cada laço

O primeiro problema encontrado foi "*Fluxo de controle dentro do loop*", ou seja é quando uma ou mais condições precisam ser satisfeitas para que realize uma operação (duas condições no loop), por exemplo, no loop dentro da função `pgain()`:

```
for( int i = k1; i < k2; i++ )
      if( is_center[i] && gl_lower[center_table[i]] > 0 )
    is_center[i] = false;
    ...
}
```

Listing 1. Fluxo de controle dentro do loop

Como se pode observar na Listagem 1, é necessário que as duas condições sejam satisfeitas para a análise condicional, assim impossibilitando a vetorização. Uma possível solução para esse problema seria definir três variáveis temporárias para os valores de `is_center[i]` atual e `gl_lower[center_table[i]] > 0` e para o resultado da comparação "&&". Definir `is_center[i]` como *true* ou *false* dependendo do valor da temporária. Além do uso de operador ternário.

Outro problema foi de "*Declaração não suportada*", que são operações não-suportadas pela vetorização, como o acesso a múltiplas variáveis não sequenciais independentes, por exemplo, também na função `pgain()`:

```
for( int p = 0; p < nproc; p++ ) {
      gl_number_of_centers_to_close += (int)work_mem[p*stride + K];
      gl_cost_of_opening_x += work_mem[p*stride+K+1];
}
```

Listing 2. Declaração não suportada

Nessa operação não suportada, as variáveis sendo agregadas são modificadas e acessadas em iterações prévias, ou seja, não são independentes. Uma maneira de contornar isso seria capturar valores de `work_mem` antes e depois somar, ao invés de adicioná-los diretamente. Da seguinte forma:

```
for( int p = 0; p < nproc; p++ ) {
      WORK_MEM = (int)wrok_mem[p*stride + K];
      WORK_MEM_1 = work_mem[p*stride+K+1];
      gl_number_of_centers_to_close += WORK_MEM;
      gl_cost_of_opening_x += WORK_MEM_1;
}
```

Listing 3. Declaração suportada

Outro problema bem comum foi "*Custo não compensa*", isso ocorre quando o compilador assume que haverá poucas iterações no laço, logo não autoriza vetorização do mesmo, por exemplo no loop interior da Listagem 4.

```
for (int i = k1; i < k2; i++) {
    if (is_center[i]) {
        double low = z;
        for (int p = 0; p < nproc; p++) {
            low += work_mem[center_table[i] + p * stride];
        }
        gl_lower[center_table[i]] = low;
        if (low > 0) {
            ++number_of_centers_to_close;
            cost_of_opening_x -= low;
        }
    }
}
```

Listing 4. Custo não compensa

Esse problema não aparenta possuir uma solução óbvia, visto que, o custo para vetorizar realmente é maior que não vetorizar o loop. Porém uma solução para o laço exterior, define-se variáveis temporárias para `is_center[i]` e `low` modificando as variáveis relevantes com uso de ternários. Para o laço interior, que define `low`, cria-se uma temporária para acumular os valores de `work_mem` e por último se define `low`.

Quando o "*Número de iterações não pôde ser computado*", ou seja o compilador não consegue identificar ou assumir um valor para o número de iterações no laço, por exemplo:

```
while((rv=pthread_mutex_trylock(&barrier->mutex)) == EBUSY){...}
```

Listing 5. Número de iterações não pôde ser computado

Uma solução para esse problema seria não ficar esperar indefinidamente dentro de um laço fixando um número máximo de iterações.

As "*variáveis voláteis*" são aquelas que podem mudar a qualquer momento, imprevisivelmente, por alguma *thread* ou *processo*. O compilador evita suposições e otimizações quando se trata de voláteis, por exemplo:

```
volatile spin_counter_t i=0;
while(barrier->is_arrival_phase && i<SPIN_COUNTER_MAX) i++;
```

Listing 6. Variáveis do tipo volátil

Uma solução para isso, seja fazer que o loop não dependa mais de variáveis voláteis, realizando outra estrategia de programação.

Por fim, o ultimo problema encontrado foi "*memory clobbers*", o que conseguimos entender que seria o fenônemo que ocorre quando os dados em memória não estão disposto de forma regular, e sim espaçados entre os intervalos da memória, dessa forma dificultando a vetorização, por exemplo:

```
while(barrier->is_arrival_phase) {
       rv = pthread_cond_wait(&barrier->cond, &barrier->mutex);
       if(rv != 0) {
         pthread_mutex_unlock(&barrier->mutex);
         return rv;
       }
```

Listing 7. Presença de “memory clobbers”

A solução poderia ser agregar dados a serem utilizados de maneira sequencial e previsível a fim de permitir vetorização.

IV. Conclusões

Portanto, percebemos que entre os problemas de vetorização encontrados, em maior parte os problemas estão nas funções `pgain()` e `pspeedy()` concentrando um total de 80% dos problemas levantados pelas *flags* de vetorização. O que reforça a análise e conclusões realizadas durante a perfilagem do programa, que apontaram essas funções como maiores possíveis gargalos do programa. Faz-se necessário, então, tentar solucionar esses problemas nessas funções específicas, buscando elevar significativamente o ganho de desempenho para esse tipo de objetivo de otimização, além de apontar o caminho a se seguir para as próximas atividade da disciplina.

Referências

[1] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.

[2] *Introduction to barriers (pthread_barrier)*. YouTube, Dec 2020. [Online]. Available: https://www.youtube.com/watch?v=_P-HYxHsVPc

[3] G. CC, “Optimize options,” 1988. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[4] A. Ltd and K. ARM Germany GmbH, “Selecting optimization options,” 2005. [Online]. Available: https://www.keil.com/support/man/docs/armclang_intro/armclang_intro_fnb1472741490155.htm