Introdução à Programação Paralela
2017/02

Trabalho Prático Parte 01: Melhora de Perfórmance em uma Multiplicação de Matrizes simples

Gustavo Borba
Belo Horizonte, September 3, 2017

1 Introdução

O presente trabalho tem o objetivo de, em sua primeira etapa, experimentar otimizações de código em um algoritmo de multiplicação de matrizes, de forma a melhorar a performance do mesmo em observancia à plataforma que será executado. Ao final desta etapa, espera-se que o código tenha um desempenho melhor, comparado ao seu original, e que esteja mais fácil de implementar sua paralelização.

2 Metodologia

De acordo com orientações descritas por BILMES[1], o código de duas das três implementações disponibilizadas foi alterado para explicitar características que podem ser utilizadas pelo otimizador do compilador.

Os arquivos com os códigos originais foram mantidos, para fins de comparação com os algoritmos alterados. Para estes últimos, uma nova linha no makefile foi criada para cada um, e na pasta do projeto pode-se verificar a presença do sufixo *borba-*, indicando uma versão alterada do arquivo.

2.1 Remover explicitamente falsas dependências

Para garantir a corretude das operações, os compiladores não assumem que, ao acessar um ponteiro que já fora acessado anteriormente, os mesmos dados estarão lá (afinal de contas, outra função pode ter acesso a esse ponteiro e alterar seu valor), de modo que as operações sobre ponteiros exigem, em sua maioria, acessos constantes à memória.
Como o programa em questão trata de três matrizes distintas, nenhuma das matrizes será alterada por uma função externa durante a execução das multiplicações. Dessa forma, é possível adicionar a cláusula **restrict** aos ponteiros recebidos pela função, que indica que apenas aqueles ponteiros ou valores diretamente derivados dele serão utilizados para acessar aquela área de memória.

```
void square_dgemm (int n,
                   double* restrict A, double* restrict B, double* restrict C)
```

O compilador saberá, dessa forma, que não é necessário realizar um fetch dos dados, à menos que haja um cache miss, pois apenas aquele ponteiro referencia a matriz em questão.

2.2 Explorando melhor os Registradores

Realizando o profiling do código original com a ferramenta **perf**, verifica-se que o tempo de acesso à memória é o principal gargalo do programa em questão. Ao verificar os *misses*, obtemos um resultado como o apresentado na Figura 1, ficam evidentes dois eventos onde se têm um custo de performance elevado

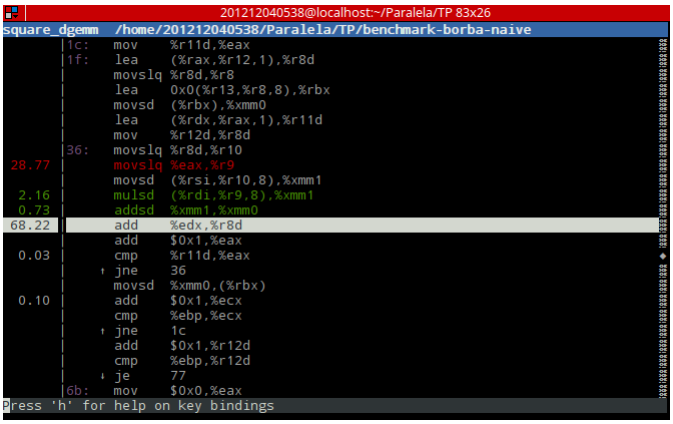

Figure 1: Profiling realizado pelo programa perf no algoritmo de multiplicação de matrizes ingênuo, em termos de *misses* totais. Percebe-se duas instruções com maior custo em performance

Analisando o código exibido, percebe-se que partes mais onerosas em termos de memória são um load e uma soma, cujo armazenamento se dá em endereço de memória.

```
void square_dgemm (int n, double* A, double* B, double* C){
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j){}
      C[i+j*n] = C[i+j*n]; // aqui
      for( int k = 0; k < n; k++ )
        C[i+j*n] += A[i+k*n] * B[k+j*n]; // aqui
    }
}
```

Para reduzir essa demanda por acessos à memória, transforma-se o ponteiro em uma variável local. Como trata-se de um valor muito utilizado, inclusive, a utilização da cláusua **register** é ainda mais eficiente. Assim, por completo, o código ingênuo se torna o seguinte:

```
void square_dgemm (int n, double* restrict A, double* restrict B, double* restrict
      C) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
      int jn = j*n;
      register double cij = C[i+jn];
      for( int k = 0; k < n; k++ ){
        double Aikn=A[i+k*n];
        double Bkjn=B[k+jn];
        cij += Aikn * Bkjn;
      }
      C[i+jn] = cij;
    }
}
```

repetindo o processo com a ferramenta **perf**, mas agora com tempo de CPU, verifica-se na Figura 2 que o gargalo de processamento ainda ocorre nestas duas operações, sendo dispensável a criação de mais variáveis temporárias para a otimização de índices, por exemplo, pois estes são pouco utilizados com relação aos anteriormente citados.

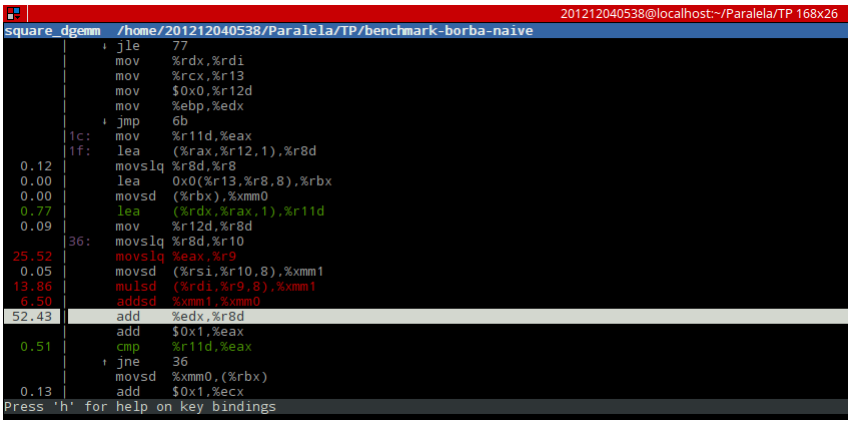

Figure 2: Profiling realizado pelo programa perf no algoritmo modificado, em termos de processamento.

3 Resultados Computacionais

Todos os testes aqui descritos foram executados na máquina Katrina, que tem como processador um Intel Haswell-EP de 6 cores (12 threads) executando a 2.4 GHz e Cache de 15 M, resultando em:

```
8 double-precision (64-bit) flops por pipeline * 6 pipelines * 2.4 GHz = 115.2 Gflops
```

Para a execução dos testes computacionais foi utilizada a função de benchmark disponibilizada para este mesmo fim. Essa função utiliza um conjunto representativo de tamanhos de matrizes (múltiplos de 32 ± 1) para executar a função de multiplicação de matrizes capturando o tempo de execução. A partir das características da máquina, a função calcula o desempenho do programa em Mflops e em porcentagem da capacidade total de processamento da máquina.

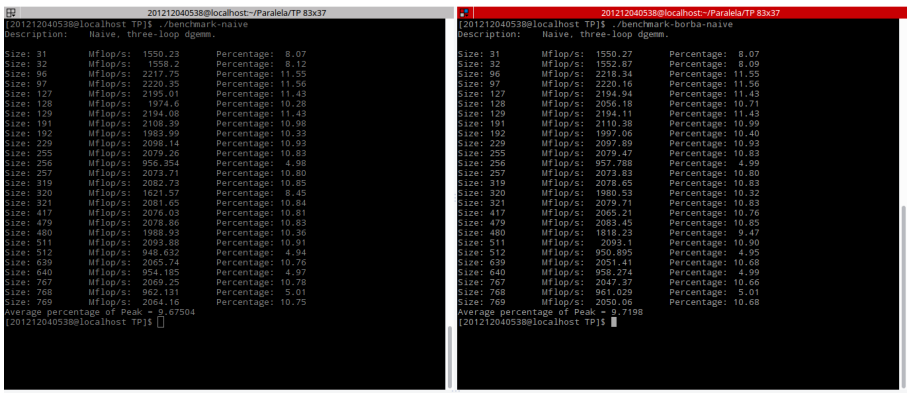

Figure 3: Execução das multiplicações ingênuas de matrizes, original (à esquerda) e modificado (à direita)

Executando os processos em sequência (para evitar a concorrência), tal como exibido na figura 5 e comparando unicamente os valores obtidos em MFlops/s foi possível montar dois gráficos, que podem ser vistos nas duas figuras abaixo. O primeiro gráfico compara a porcentagem de uso da capacidade total da CPU do servidor Katrina, enquanto o segundo gráfico explora as diferenças em percentual dos dois algoritmos.

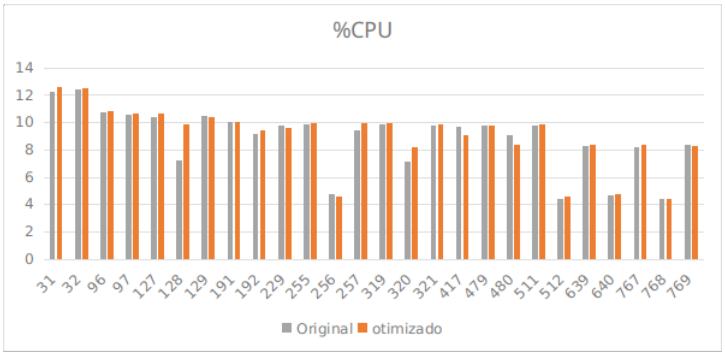

Figure 4: Rendimento em porcentagem de Capacidade total da CPU para os algoritmos

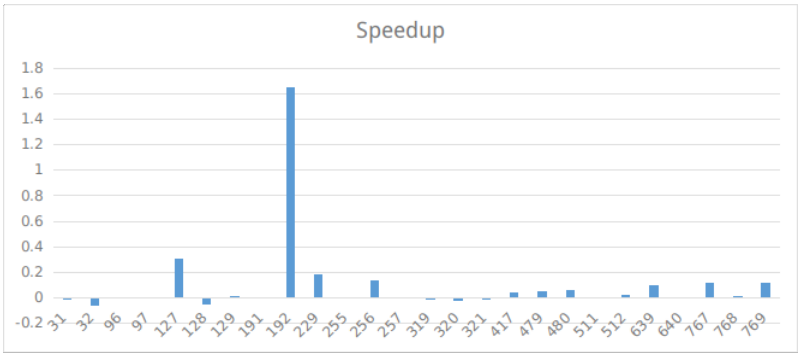

Figure 5: Diferença percentual entre o algoritmo utilizado e o algoritmo

Percebe-se uma grande otimização para o tamanho de matrizes de 192x192. Esse valor é um divisor do tamanho da cache, o que indica que armazenar o valor mais utilizado em um registrador pode realmente valer a pena:

$$15M = 1024 \times 15 = 15360.$$

$$\frac{15360}{192} = 80\text{páginas}$$

Utilizando a mesma estrutura de testes, agora compara-se o código escrito com otimizações e seu código original compilado com as diretivas *-O3 fstrict-aliasing -std=gnu99*, obtivemos o gráfico da figura 6, que representa o ganho (em porcentagem) da otimização realizada manualmente ante à otimização do compilador.

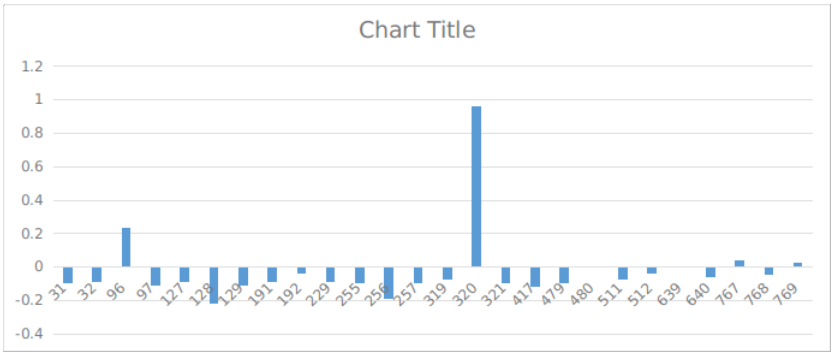

Figure 6: Resultados do ganho da otimização escrita, em comparação com a realizada pelo compilador

Nota-se que os resultados do otimizador do compilador são ligeiramente melhores (valores negativos) enquanto as otimizações manuais se mostraram mais efetivas nos tamanhos de matrizes 96 e 320 (chegando a 1% a mais de utilização da capacidade total da CPU). Tal resultado pode ser explicado pelas características do processador (que tem um processador de 3.20 GHz) em conjunto com a otimização da gerência de memória, utilizando um registrador para armazenar o valor mais comumente utilizado, evitando cache misses e aproveitando melhor o tempo para realizar operações de ponto flutuante.

4 Conclusão

As otimizações realizadas foram apenas ligeiramente eficientes, chegando próximas ao otimizador do compilador, mas perdendo um pouco para ele. Estima-se, no entanto, que estas otimizações ajudem a melhorar significantemente o paralelismo que será futuramente feito.
Os próximos passos envolvem Loop Unrolling, e a pesquisa e implementação de um método mais eficiente de multiplicação de matrizes.

References

[1] BILMES, J. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology
Disponível em *https:people.eecs.berkeley.edu~krstepapersphipac_ics97.pdf*
Acesso em 2 de Setembro de 2017