Capítulo 6

Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código

Sherlon Almeida da Silva[1] – sherlonalmeida@alunos.unipampa.edu.br
Matheus S. Serpa[2] – msserpa@inf.ufrgs.br
Claudio Schepke[3] – claudioschepke@unipampa.edu.br

Resumo

Obter desempenho de uma aplicação exige conhecer profundamente a arquitetura dos computadores e utilizar boas práticas e técnicas de programação, a fim de explorar todos os recursos disponíveis. A análise de desempenho de uma aplicação pode ser consolidada através de verificações teóricas, testes manuais ou através de uma série de execuções utilizando ferramentas de análise de dados e execução. Tradicionalmente as funções ou trechos de código que demandam mais tempo de execução são modificadas a fim de melhor aproveitar a arquitetura e/ou utilizar princípios de programação paralela. Existem várias ferramentas que possibilitam avaliar o comportamento de programas sequenciais e paralelos, tais como: Gprof, Perf, Valgrind, Intel Vtune e Intel Performance Counter Monitor (PCM). Nesse contexto, a proposta deste minicurso é apresentar técnicas de otimização e fazer uso de ferramentas de análise de desempenho, para ter medidas precisas

[1]Sherlon Almeida da Silva é estudante de Bacharelado em Ciência da Computação pela Universidade Federal do Pampa, campus Alegrete/RS. Atualmente é bolsista da Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS). Sua área de pesquisa é Computação de Alto Desempenho com ênfase em análise e otimização de algoritmos. Participa do grupo de pesquisa Laboratório de Estudos Avançados em Computação (LEA) da UNIPAMPA.

[2]Matheus da Silva Serpa é atualmente bolsista da Intel Corporation e Mestrando no Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (UFRGS). É bacharel em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA), tendo sido bolsista de iniciação científica da FAPERGS por três anos, e recebido o Prêmio Destaques UNIPAMPA 2015 e o Prêmio Aluno Destaque da SBC (2016). Suas principais áreas de pesquisa são Arquitetura de Computadores e Computação de Alto Desempenho.

[3]Claudio Schepke possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2005) e mestrado (2007) e doutorado (2012) em Computação pela Universidade Federal do Rio Grande do Sul. Atualmente é professor adjunto da Universidade Federal do Pampa, campus Alegrete/RS, onde ministra disciplinas de programação e coordena projetos envolvendo aplicações e interfaces de programação paralelas.

do impacto dessas otimizações no desempenho das aplicações sequenciais e paralelas. O uso de técnicas de desenrolamento de laços, acessos em blocos (tiling) para maximizar o uso da cache e reordenação de operações são algumas das opções que serão discutidas e apresentadas de forma teórica e prática no minicurso. Além disso, as otimizações serão validadas através de métricas retiradas das ferramentas Perf e PCM.

6.1. INTRODUÇÃO

A utilização da computação para resolução de problemas científicos é constante. Isso é mais expressivo ainda quando se trata de cálculos que lidam com grandes volumes de dados, como é o caso de simulações computacionais de fenômenos da natureza. Profissionais de diversas áreas usam a computação para fundamentar suas pesquisas científicas. A agilidade no retorno de resultados durante a execução das aplicações é um dos fatores que possibilita o avanço científico atual.

Não é interessante para a ciência investir na previsão de um fenômeno natural sabendo-se que as respostas sobre ele serão obtidas após o seu acontecimento. Não haveria utilidade neste investimento para casos de terremotos ou furacões, por exemplo. Por causa disso o alto desempenho na execução de aplicações científicas é necessário. Quanto maior for a capacidade de processamento da máquina utilizada juntamente com boas práticas de programação, mais rápidos e precisos poderão ser os resultados retornados.

Tendo por objetivo suprir as necessidades de velocidade na execução de aplicações foram desenvolvidas técnicas para o aumento do desempenho do *hardware* [Clapp 2015, Clapp et al. 2010]. As arquiteturas sequenciais e paralelas são fundamentais para a agilidade dos computadores atuais. Enquanto a primeira executa tarefas em sequência em um único núcleo de processamento, a segunda pode executar as tarefas simultaneamente em vários núcleos integrados ao processador. Estas máquinas com diversos núcleos são chamadas de *multi-core*. A utilização do paralelismo em máquinas *multi-core* possibilita executar uma aplicação mais rapidamente, pois mais de uma tarefa pode estar executando simultaneamente.

Além da utilização de vários núcleos de processamento é importante o uso adequado da memória *cache*, que é uma memória rápida localizada próxima ao processador. O objetivo desta memória é manter os dados frequentemente utilizados próximos à CPU, para assim reduzir o tempo de acesso à memória principal que possui alta latência. A presença de vários núcleos e memória *cache* em uma máquina só garantirão desempenho se forem explorados da maneira correta. Para isto existem diversas técnicas de programação e de análise de desempenho que possibilitam atingir, a partir de otimizações, um alto ganho no desempenho nas aplicações.

O objetivo deste minicurso é apresentar o método de construção de uma aplicação desde o desenvolvimento, análise e otimização da mesma. Com este intuito serão abordadas técnicas sequenciais de otimização, tais como: *loop interchange*, *loop unrolling* e *loop tiling*, bem como execuções paralelas em múltiplas *threads*. As respostas a respeito do ganho de desempenho de uma aplicação serão obtidas a partir da coleta de dados e da análise de informações proporcionadas pelas ferramentas Gprof, Perf, e Intel PCM.

Este documento está organizado da seguinte forma. Na Seção 6.2 são apresenta-

das brevemente as maneiras de se otimizar e analisar aplicações. A Seção 6.3 expõe as técnicas de otimização e ganho de desempenho mencionadas na seção anterior. A Seção 6.4 introduz às ferramentas utilizadas neste trabalho, indicando suas funcionalidades e como usá-las. A aplicação das técnicas e ferramentas abordadas neste minicurso estão na Seção 6.5. Por fim a Seção 6.6 apresenta as considerações finais deste trabalho.

6.2. IMPLEMENTANDO APLICAÇÕES ÁGEIS

O desenvolvimento de uma aplicação eficiente abrange várias etapas. Estas etapas serão mostradas neste minicurso com o intuito de proporcionar o conhecimento de diferentes técnicas e abordagens para ganho de desempenho em aplicações. Desde a fase de implementação de uma aplicação até a coleta de dados para a análise de desempenho são necessários diversos testes. Por exemplo, executar um algoritmo dezenas de vezes para obter a média aritmética, o desvio padrão e o número de acessos à memória *cache* são formas de avaliar o comportamento de uma aplicação.

O programador pode executar um algoritmo e registrar o seu resultado facilmente exportando o resultado em um arquivo de texto. Porém, a análise dos dados e o entendimento do fluxo do programa e do ganho obtido podem não ser tão precisos sem o uso de ferramentas específicas para tal fim. O que se faz para obter métricas de processamento é executar uma dada aplicação várias vezes e comparar os dados retornados através das ferramentas de análise de desempenho. O uso de técnicas de otimização são fundamentais para melhorar o desempenho de aplicações, por isso a análise bem feita dos dados é muito importante.

O conhecimento do comportamento de um programa durante sua execução é essencial para aplicar otimizações. Por exemplo, as funções que demandam mais tempo devem ser modificadas para melhor aproveitamento de memória *cache* e para utilizar os princípios de programação paralela nas arquiteturas *multi-core*. Existem várias ferramentas que possibilitam tais avaliações de desempenho, como: AMD CodeAnalyst, Intel Vtune, Sysprof, Oprofile, Valgrind, Gprof, Perf, Intel PCM, entre outras.

Quando o programador domina o comportamento da execução de uma dada aplicação em determinada arquitetura e também sabe o custo em tempo e processamento de cada função de seu código, sua programação pode explorar tal aspecto da arquitetura utilizada para melhorar o desempenho do código. Usando métodos de análise da ferramenta Gprof, por exemplo, nota-se que a multiplicação é a função mais custosa de um algoritmo de multiplicação de matrizes. Sabendo disso, pode-se programar utilizando os conceitos de princípio de localidade na memória *cache* e de paralelismo com execuções em vários núcleos, visando aumentar o desempenho da aplicação.

É possível aumentar o desempenho de uma aplicação quando se identifica a função do algoritmo que é mais custosa. A partir disso, é possível coletar amostras de acertos e faltas em *cache* em tempo de execução com as ferramentas Perf e Intel PCM. Desta forma, busca-se compreender estatisticamente a performance da aplicação na arquitetura utilizada. Estas ferramentas de análise auxiliam o programador na avaliação de seu código e na otimização do mesmo, consequentemente tais ferramentas auxiliam diretamente no ganho de desempenho.

As técnicas apresentadas neste minicurso encontram-se disponíveis no github. Estão disponíveis vários códigos, em linguagem de programação C, baseados em uma aplicação de multiplicação de matrizes. Estes códigos serão otimizados e analizados gradativamente durante o minicurso.

Download em: `https://github.com/msserpa/otimizacaoErad2017`

6.3. TÉCNICAS DE OTIMIZAÇÃO UTILIZADAS

A busca de instruções e dados pelo processador está relacionada à comunicação entre a CPU e a memória principal. A alta latência de acesso à memória principal é um dos limitadores de desempenho. A utilização da memória *cache* serve para minimizar o impacto dessa latência, uma vez que ela armazena dados frequentemente utilizados pelo processador baseando-se no princípio da localidade [Lee et al. 2010].

A Figura 6.1 ilustra o objetivo da utilização da memória *cache*. Quando uma aplicação é executada suas tarefas são carregadas para a memória principal, e aquelas instruções e dados mais utilizados mantém-se em *cache* para evitar a busca de maior latência na memória principal. A CPU busca primeiramente nos níveis da *cache* partindo do nível L1 até a memória principal, quanto mais próximo à CPU os dados se encontrarem mais rápida será a execução. Em uma arquitetura com três níveis de *cache* (L1, L2 e L3) deseja-se manter os dados o maior tempo possível em L1 para garantir maior velocidade na execução.

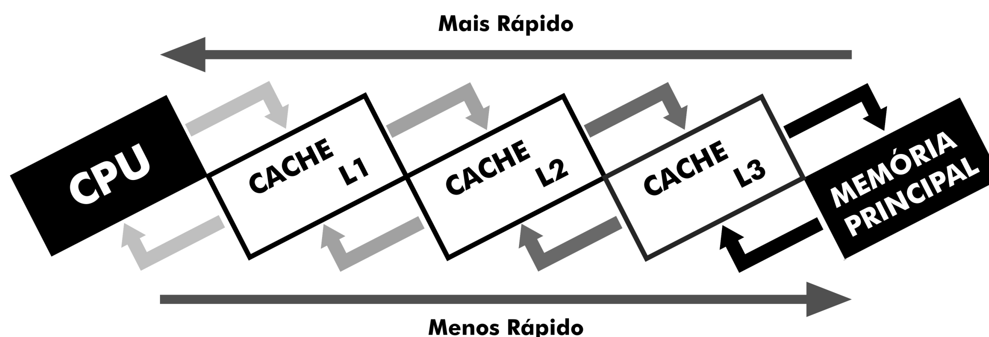


Figura 6.1. Memória *cache*.

Entendendo o objetivo e o funcionamento de uma memória *cache* pode-se aplicar técnicas para fazer aquilo que aproveita melhor o seu uso. Com a finalidade de potencializar a eficiência do algoritmo existem técnicas de *loop* que possibilitam o uso eficiente de memória *cache*, tais como *loop interchange*, *loop unrolling* e *loop tiling*. Tais técnicas serão abordadas nas Subseções 6.3.1, 6.3.2 e 6.3.3.

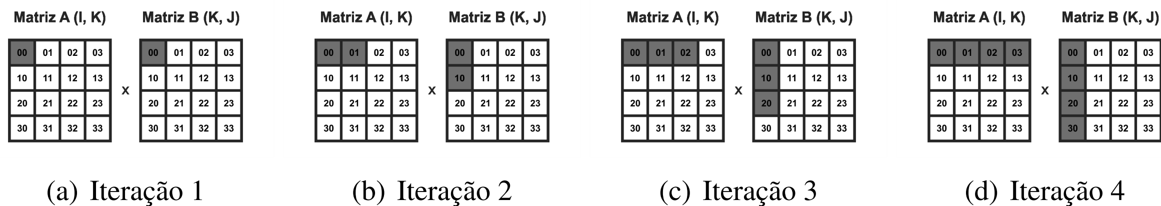
6.3.1. LOOP INTERCHANGE

A técnica de *loop interchange* trabalha nos laços de repetição das aplicações visando reordená-los para melhor aproveitar os dados em *cache*. Ao alterar a ordem dos laços de repetição aninhados o programador faz com que os índices controlados por este laço sejam percorridos de forma diferente da original. Em um algoritmo de multiplicação de matrizes convencional sem o uso desta técnica há a atualização dos elementos da matriz A e da matriz B a cada iteração, tornando a execução mais demorada pois lê os dados

constantemente. Como pode-se ver no Algoritmo 1 o código possui três laços de repetição, sendo o primeiro **i**, o segundo **j** e o terceiro **k**. A execução deste algoritmo é ilustrada na figura 6.2.

```
1 for(i = 0; i < SIZE; i++)
2   for(j = 0; j < SIZE; j++)
3     for(k = 0; k < SIZE; k++){
4       C(i, j) += A(i, k) * B(k, j);
5     }
```

Algoritmo 1: Versão *Convencional*



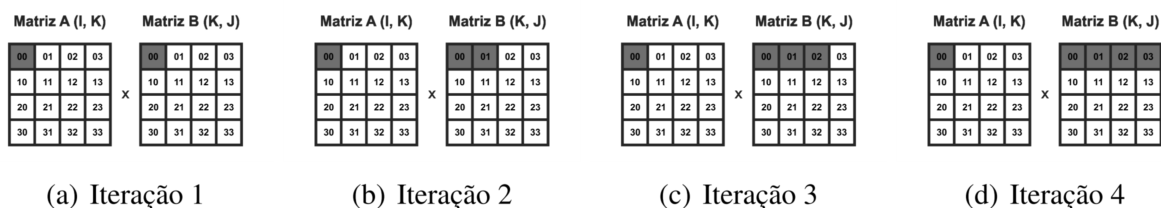
(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

Figura 6.2. Matriz Convencional - Laços I J K.

A técnica de *loop interchange* é mostrada no Algoritmo 2, como pode-se ver o algoritmo com esta técnica inverte a ordem dos laços de repetição. Os laços estão ordenados assim, o primeiro **i**, o segundo **k** e o terceiro **j**. Esta maneira de percorrer os laços de repetição acarreta em uma melhora no aproveitamento dos dados em cache pois mantém por mais tempo em memória *cache* cada elemento da matriz A e multiplica os elementos da matriz B na ordem em que estão endereçadas na memória, aproveitando melhor a localidade espacial. A função destes laços na multiplicação pode ser observada na Figura 6.3, onde nota-se que o elemento da matriz A só é incrementado após ter sido multiplicado com todos seus correspondentes na matriz B.

```
1 for(i = 0; i < SIZE; i++)
2   for(k = 0; k < SIZE; k++)
3     for(j = 0; j < SIZE; j++){
4       C(i, j) += A(i, k) * B(k, j);
5     }
```

Algoritmo 2: Versão *Interchange*



(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

Figura 6.3. Matriz Convencional - *Loop Interchange* - Laços I K J.

Esta técnica permite um alto ganho de desempenho pois, como mostram as Figuras 6.2 e 6.3, os dados da matriz A são carregados uma vez e são utilizados para varrer

a matriz B, diferente da versão convencional que atualiza todos os dados por iteração. Operando desta forma os dados da matriz A são melhor aproveitados em *cache* do que a execução convencional.

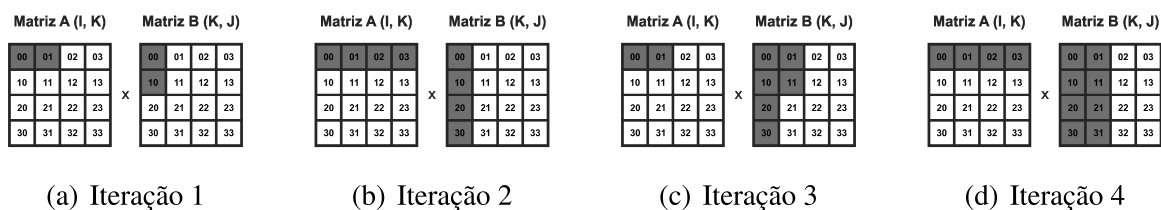
6.3.2. LOOP UNROLLING

A técnica de *loop unrolling* utiliza mais de um incremento dentro do laço da multiplicação do algoritmo de multiplicação de matrizes. No laço mais interno a multiplicação é realizada baseando-se no número de desenrolamentos que o programador define. O Algoritmo 3 mostra o código fonte desta técnica. Como observa-se, o algoritmo desenrola várias vezes por iteração com a replicação do trecho de código e a inserção da variável k+1 para carregar o próximo elemento da matriz.

```
1 for(i = 0; i < SIZE; i++)
2   for(j = 0; j < SIZE; j++)
3     for(k = 0; k < SIZE; k+=2){
4       C(i, j) += A(i, k) * B(k, j);
5       C(i, j) += A(i, k+1) * B(k+1, j);
6     }
```

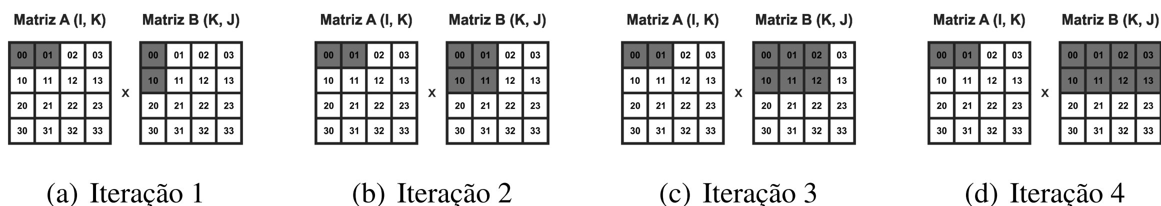
Algoritmo 3: Versão *Unrolling*

Esta técnica agiliza a execução pois tem como objetivo melhorar o desempenho do programa reduzindo o número de saltos (*branches*) através da alteração do número de somas por iteração. O comportamento desta técnica na multiplicação pode ser observado nas Figuras 6.4 e 6.5, onde cada iteração executa 2 elementos.



(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

Figura 6.4. Matriz Convencional - *Loop Unrolling* - Laços I J K.



(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

Figura 6.5. Matriz Convencional - *Loop Interchange* - *Loop Unrolling* - Laços I K J.

No algoritmo convencional para percorrer os elementos das matrizes A e B eram incrementadas de um em um as variáveis, assim ocorria a leitura para a multiplicação, a multiplicação entre os elementos da matriz A e B e o resultado era armazenado na matriz C. Agora não acontece de um em um, acontece a partir de um valor que o programador define. Por exemplo, na soma de vetores de 1000 elementos haveriam 1000 repetições no

método tradicional com um incremento por iteração, mas caso o programador utilizasse a técnica de *loop unrolling* e definisse o *unroll* como 5, haveriam apenas 200 repetições com 5 somas por iteração, reduzindo os *branches*. Veja nos Algoritmos 4 e 5 como ficam os códigos.

```
1 for(i = 0; i < 1000; i++){
2  vetor_C[i] = vetor_A[i] + vetor_B[i];
3 }
```

Algoritmo 4: Soma de vetores SEM *Loop Unrolling*

```
1 for(i = 0; i < 200; i+=5){
2  vetor_C[i] = vetor_A[i] + vetor_B[i];
3  vetor_C[i+1] = vetor_A[i+1] + vetor_B[i+1];
4  vetor_C[i+2] = vetor_A[i+2] + vetor_B[i+2];
5  vetor_C[i+3] = vetor_A[i+3] + vetor_B[i+3];
6  vetor_C[i+4] = vetor_A[i+4] + vetor_B[i+4];
7 }
```

Algoritmo 5: Soma de vetores COM *Loop Unrolling*

6.3.3. LOOP TILING

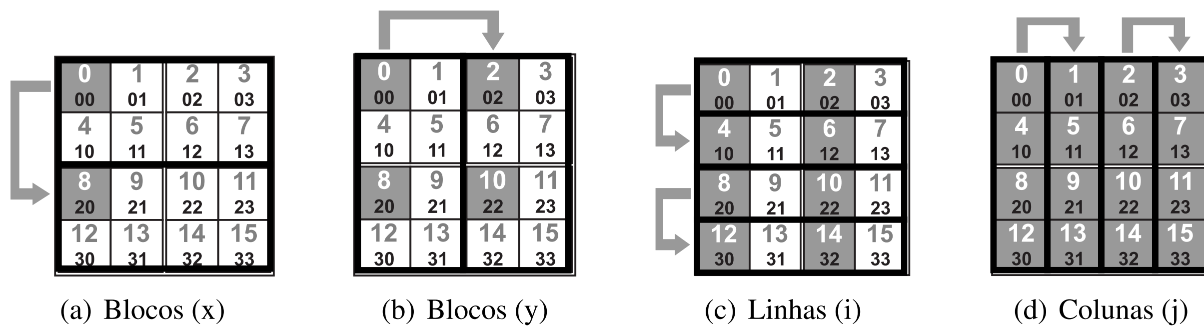
A técnica *loop tiling* fragmenta a matriz inicial em diferentes matrizes que são chamadas de blocos. Este método pode ser interpretado como "Matriz de Matrizes". Estes blocos originados da matriz inicial podem ser computados separadamente como uma multiplicação convencional de matrizes, o que difere é que ocorre uma sincronização dos dados dos blocos no final do cálculo.

No Algoritmo 6 o deslocamento das operações ocorre conforme ilustra a Figura 6.6. As variáveis **x** e **y** percorrem os blocos linhas e colunas. E dentro destes blocos as variáveis **i** e **j** percorrem as linhas e colunas.

```
1 for(x = 0; x < size; x += TILE_SIZE)
2   for(y = 0; y < size; y += TILE_SIZE)
3     for(z = 0; z < size; z += TILE_SIZE)
4       for(i = x; i < x + TILE_SIZE; i++)
5         for(j = y; j < y + TILE_SIZE; j++)
6           for(k = z; k < z + TILE_SIZE; k++){
7             C(i, j) += A(i, k) * B(k, j);
8           }
```

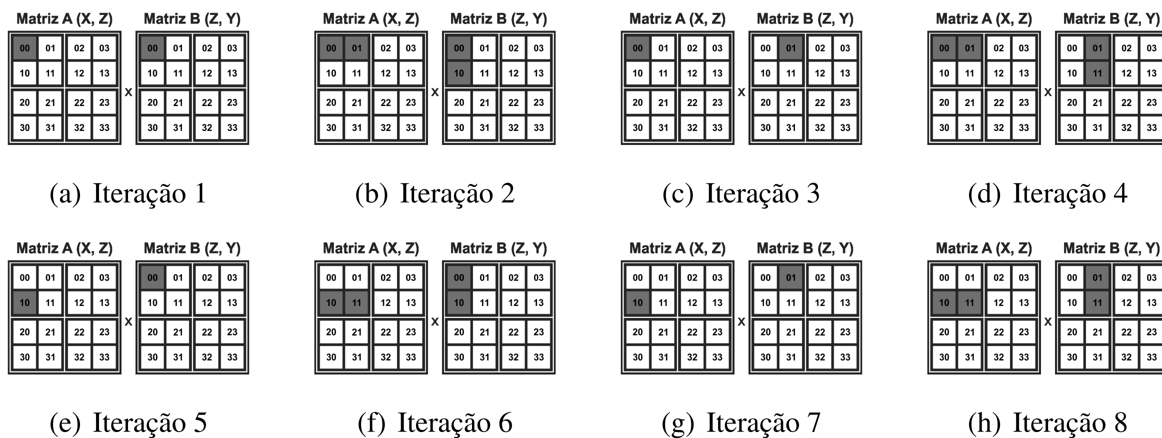
Algoritmo 6: Versão *Tiling*

Note que na Figura 6.7 o desenrolamento dos elementos ocorre calculando os blocos como elementos únicos, após definir os blocos a serem multiplicados os elementos internos a cada um desses blocos são multiplicados entre si. Exatamente como uma multiplicação de matrizes dentro de uma multiplicação de matrizes, sendo a primeira para definir quais blocos serão multiplicados e a segunda a verdadeira multiplicação entre os elementos destes blocos.



(a) Blocos (x) (b) Blocos (y) (c) Linhas (i) (d) Colunas (j)

Figura 6.6. Deslocamento dos elementos da matriz.



(a) Iteração 1 (b) Iteração 2 (c) Iteração 3 (d) Iteração 4

(e) Iteração 5 (f) Iteração 6 (g) Iteração 7 (h) Iteração 8

Figura 6.7. *Loop Tiling* - Laços I J K X Y Z.

Esta técnica é eficiente pois armazena toda a submatriz em *cache*, reduzindo o acesso a memória principal [Satish et al. 2012]. Para matrizes que não podem ser armazenadas totalmente em *cache*, esta técnica possibilita ganho de desempenho. Entretanto, matrizes de ordem inferior a 512 não ganham significativamente com a *Tiling* pois elas cabem totalmente em *cache* L3 independente de serem particionadas em blocos.

6.3.4. ÚLTIMA ETAPA DE OTIMIZAÇÃO

Ao aplicar todas as técnicas apresentadas é possível ter um alto ganho de desempenho pois aumenta a eficácia no uso da memória *cache*. A partir da compreensão do funcionamento de cada técnica utilizada é possível então explorar o paralelismo na aplicação.

Deve-se conhecer a máquina utilizada, seu número de núcleos físicos e lógicos e então aplicar alguma interface de programação de aplicações paralelas no código para gerenciar o paralelismo.

6.4. DESCRIÇÃO DAS FERRAMENTAS UTILIZADAS

Foram apresentadas as técnicas para otimização das aplicações, porém é muito importante analisar de forma eficiente com algumas ferramentas. A análise das execuções será validada utilizando as ferramentas Gprof, Perf e Intel PCM [Intel 2012].

Haverá uma explicação sobre cada uma das ferramentas e seus principais comandos. Cada ferramenta possui recursos diferentes de análise de desempenho. Com o Gprof

serão destacadas as funções mais custosas ao tempo de execução da aplicação. Serão aplicadas as técnicas de otimização e as ferramentas Perf e PCM realizarão a coleta de dados referentes à memória *cache*, entre outras.

6.4.1. GPROF

A ferramenta Gprof está vinculada ao gcc, seu criador foi Jay Fenlason [Chaves 2010]. Suas funcionalidades permitem analisar o desempenho de um algoritmo e mostrar os resultados detalhados de cada função. Esta ferramenta possibilita ao programador conhecer o número de funções que o código possui, quantas vezes cada função é chamada e o tempo gasto durante a execução da função.

Ferramentas como o Gprof permitem localizar trechos de código que são frequentemente utilizados. Desta forma é possível obter uma análise de tempo de execução em cada função do programa. A utilização do Gprof é fundamental na primeira etapa de análise de uma aplicação pois conhecer as funções mais custosas permite ao programador saber qual trecho de código é preciso otimizar.

Primeiramente entre na pasta onde há um programa .c ou .cpp. Para usar o Gprof serão utilizados dois comandos, basicamente. Agora basta digitar no terminal os comandos encontrados no Algoritmo 7:

```
1 gcc -pg -o programa nome_do_programa.c
2 ./programa
3 gprof programa //Gera o resultado completo
4 gprof -p programa //Gera o resultado resumido
5 gprof -q programa //Gera o resultado e separa as funcoes
```

Algoritmo 7: Execução do Gprof

Ao compilar o programa com a flag de compilação -pg esta flag habilita o compilador a coletar as medições durante a execução do código. Após a compilação e execução do programa será gerado um arquivo com nome default gmon.out que deverá ser interpretado pelo Gprof.

Ao executar é interessante registrar os dados coletados para posterior visualização, caso tenha muitos arquivos e funções no código utilizado. Para isso, pode-se armazenar tais dados em um arquivo de texto. Basta utilizar um redirecionamento de saída para o arquivo .txt desejado. Exemplo:

gprof programa » saida_do_programa.txt *//gera arquivo de saída*

Existem outras *flags* que podem ser utilizadas, -z mostra todas as funções mesmo não sendo utilizadas, -c mostra quais rotinas nunca foram chamadas, -b serve para tirar linhas que explicam o significado de todos os campos da tabela, -a suprime a impressão de funções estaticamente declaradas (privadas) e suas filhas e atribui o tempo gasto à função carregada anteriormente.

A utilização do Gprof requer que seu código esteja particionado em funções. Então caso tenha implementado uma aplicação com sua estrutura interna à função main, separe-a em funções antes de utilizar o Gprof. Executando um programa de Multiplicação

de Matrizes se tem o seguinte passo a passo e resultados, veja a seguir nas Figuras 6.8, 6.9 e 6.10, respectivamente:

Primeiro: compila-se o programa usando a *flag* -pg para permitir a coleta.

```
shevs@shevs-VirtualBox:~/Área de Trabalho$ gcc -pg -o programa 02_Matriz_Convenc
ional_Funcoes_Separadas.c -fopenmp
```

Figura 6.8. Passo 1 - Compilar.

Segundo: executa-se o programa passando os parâmetros (caso tenha parâmetro).

```
shevs@shevs-VirtualBox:~/Área de Trabalho$ ./programa 2048 2048
```

Figura 6.9. Passo 2 - Executar programa.

Terceiro: executa-se o Gprof redirecionando a saída para um arquivo.

```
shevs@shevs-VirtualBox:~/Área de Trabalho$ gprof programa >> saida_gprof.txt
```

Figura 6.10. Passo 3 - Executar Gprof.

O arquivo saida_gprof.txt contém o seguinte conteúdo:

```
3 Each sample counts as 0.01 seconds.
4   %   cumulative   self              self     total
5  time   seconds   seconds    calls   s/call   s/call  name
6 100.47    148.36   148.36        1   148.36   148.36  multiplica
7   0.02    148.39     0.03        2     0.02     0.02  inicializa
8   0.00    148.39     0.00        3     0.00     0.00  aloca
9   0.00    148.39     0.00        3     0.00     0.00  desaloca
```

Figura 6.11. Dados gerados após a execução do Gprof.

Os campos na parte superior da Figura 6.11 descrevem as funcionalidades de cada coluna de dados, as quais são explicadas na Tabela 6.1.

6.4.2. PERF

Esta ferramenta permite compreender o uso da CPU através da análise de *hardware* e possibilita ao programador destacar o número de ciclos de *clock*, acessos aos níveis de *cache*, análise de tempo de execução, entre outras funcionalidades.

Para instalar a ferramenta abra o terminal utilizando o comando ***ctrl+alt+t*** e digite o comando a seguir:

sudo apt-get install linux-tools-common *//baixa o perf da lista de repositórios*

Tabela 6.1. Significado das colunas no Gprof

CAMPO	SIGNIFICADO
% time	Percentual de Tempo.
cumulative seconds	Tempo gasto pela função mais as demais funções acima na tabela.
self seconds	Tempo gasto pela função.
calls	Número de chamadas da função.
self ms/ call	Tempo em milissegundo para chamar a função.
total ms/ call	Tempo em milissegundo para chamar a função e seus descendentes.
name	Nome da função.

O sistema poderá pedir a instalação de outras ferramentas, instale-as usando o mesmo comando anterior ***sudo apt-get install nome_do_programa*** e pronto, o Perf já estará pronto para uso após o download.

Abaixo encontram-se alguns comandos utilizados no Perf para análise de execução. Serão mostrados alguns comandos para serem testados, porém o leitor está livre para buscar os demais comandos que possam ser úteis em suas pesquisas.

Lista de Eventos:

Os eventos disponíveis na máquina utilizada são mostrados com o comando ***perf list***, isso irá mostrar quais eventos poderão ser utilizados para análise de execução, os quais poderão ser manipulados para atender as necessidades de análise de cada pesquisa.

perf list *//Lista todos eventos atuais conhecidos*

Lista dos Eventos Pré-Definidos: A seguir encontram-se os eventos que podem ser utilizados para análise de execução, sendo eles: ***Hardware Event, Software Event, Hardware Cache Event*** *e* ***Kernel PMU Event***.

Hardware Event: cpu-cycles OR cycles, instructions, cache-references, cache-misses, branch-instructions OR branches, branch-misses, stalled-cycles-frontend OR idle-cycles-frontend, stalled-cycles-backend OR idle-cycles-backend, ref-cycles.

Software Event: cpu-clock, task-clock, page-faults OR faults, cpu-migrations OR migrations, minor-faults, major-faults, alignment-faults, context-switches OR cs, dum my, emulation-faults.

Hardware Cache Event: L1-dcache-loads, L1-dcache-load-misses, L1-dcache-stores, L1-dcache-store-misses, L1-dcache-prefetches, L1-dcache-prefetch-misses, L1-icache-loads, L1-icache-load-misses, LLC-loads, LLC-load-misses, LLC-stores, LLC-store-misses, LLC-prefetches, LLC-prefetch-misses, iTLB-loads, iTLB-load-misses, bra nch-loads, branch-load-misses, node-loads, node-load-misses, node-stores, node-store-misses, node-prefetches, node-prefetch-misses, dTLB-loads, dTLB-stores, dTLB-load-misses, dTLB-store-misses.

Kernel PMU Event: branch-instructions OR cpu/branch-instructions/, branch-

misses OR cpu/branch-misses/, cache-misses OR cpu/cache-misses/, cache-references OR cpu/cache-references/, cpu-cycles OR cpu/cpu-cycles/, instructions OR cpu/instructio ns/, mem-loads OR cpu/mem-loads/, ref-cycles OR cpu/ref-cycles/, stalled-cycles-backend OR cpu/stalled-cycles-backend/, stalled-cycles-frontend OR cpu/stalled-cycles-frontend/, uncore/clockticks/, uncore/qhl_request_ioh_reads/, uncore/qhl_request_ioh_writes/, uncore/qhl_request_local_reads/, uncore/qhl_request_local_writes/, uncore/qhl_request_remote_reads/, uncore/qhl_request_remote_writes/, uncore/qmc_normal_reads_any/, uncore/qmc_writes_full_any/.

A utilização dos comandos do Perf baseiam-se na declaração da Ação, do Evento, do Escopo e do Comando seguido dos seus Parâmetros de entrada. O comando ***perf list*** mostra os eventos disponíveis na máquina utilizada, o ***perf stat*** gerencia os contadores de eventos, o ***perf record*** captura as informações e salva em ***perf.data***, onde o ***perf report*** analisa e retorna o resultado da análise. O ***perf script*** organiza os dados para visualização.

EXEMPLOS:

A partir dos eventos disponíveis é possível utilizar os comandos de análise. Alguns comandos frequentemente utilizados do Perf estão dispostos abaixo:

Coleta de Loads, Misses e Stores da Cache L1:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores executável parâmetros

Coleta de Loads e Misses do último nível de Cache:
perf stat -e LLC-loads,LLC-load-misses executável parâmetros

É importante usar no máximo 2 eventos por coleta para não ultrapassar o número de contadores de *hardware* disponíveis na arquitetura utilizada.

Por exemplo, pode-se coletar insformações sobre o nível L1 da memória *cache* com o seguinte comando:

perf stat -e L1-dcache-loads,L1-dcache-load-misses ./programa parâmetros

Ao executar o comando como na Figura 6.12 a saída da execução é redirecionada para um arquivo de texto. O arquivo de texto gerado contém as informações mostradas na Figura 6.13. Estas informações geradas são o número de leituras e faltas em *cache* L1, onde também há o percentual de *misses* em relação ao total de *loads*. Com esses dados pode-se analisar as execuções das diferentes técnicas de otimização apresentadas para observar o impacto de cada otimização.

```
sherlon@u02s065908:~$ perf stat -e L1-dcache-loads,L1-dcache-load-misses ./progr
ama 128 128 &> saida.txt
```

Figura 6.12. Executando o Perf.

```
Performance counter stats for './programa 128 128':

        6.673.771      L1-dcache-loads
          164.701      L1-dcache-load-misses     #    2,47% of all L1-dcache hits

      0,007598565 seconds time elapsed
```

Figura 6.13. Saída gerada após a execução do Perf.

6.4.3. INTEL PERFORMANCE COUNTER MONITOR (PCM)

Durante a execução de uma aplicação é importante obter detalhes sobre o processamento na máquina. Para coletar tais dados é necessário um software sofisticado com esta proposta. Para isto basta baixar a ferramenta Intel *Performance Counter Monitor* (PCM) que se encontra disponível em: <https://github.com/opcm/pcm>.

Esta ferramenta possibilita que o programador obtenha detalhes físicos da máquina utilizada e mostra detalhadamente dados sobre a execução, como o acesso aos níveis de *cache*, memória utilizada, tempo gasto, taxas de acerto, energia, entre outras. É uma excelente ferramenta para análise e coleta de dados durante uma execução.

Para instalar o Intel PCM primeiramente baixe a aplicação e descompacte o arquivo. A partir disso entre na pasta descompactada e comente a linha 16 do arquivo ***Makefile*** para desabilitar a coleta com o Perf e deixar padrão a coleta direto dos contadores de *hardware*. Após feito isso abra o terminal na pasta descompactada e digite ***make*** para instalar a ferramenta. Pronto!

Para executar basta digitar o seguinte comando no terminal:

sudo caminho1/./pcm.x - - caminho2/seu_executavel os_parâmetros

O ***caminho1*** a ser inserido é o endereço do diretório do Intel PCM e o ***caminho2*** é o endereço do diretório do executável do programa. Veja a execução na Figura 6.14 e o arquivo gerado na Figura 6.15.

```
sherlon@u02s065908:~$ sudo ~/pcm-master/pcm.x -- ~/programa 128 128
```

Figura 6.14. Executando o Intel PCM.

```
Number of physical cores: 16
Number of logical cores: 32
Number of online logical cores: 32
Threads (logical cores) per physical core: 2
Num sockets: 2
Physical cores per socket: 8
Core PMU (perfmon) version: 3
Number of core PMU generic (programmable) counters: 4
Width of generic (programmable) counters: 48 bits
Number of core PMU fixed counters: 3
Width of fixed counters: 48 bits
Nominal core frequency: 2000000000 Hz
```

Figura 6.15. Saída gerada após a execução do Intel PCM.

Além das informações de *hardware* disponíveis sobre a arquitetura utilizada há o detalhamento da execução. A partir destes dados gerados pode-se analisar mais profundamente os aspectos de memória *cache*, entre outras.

6.5. HANDS-ON - OTIMIZANDO E ANALISANDO UMA APLICAÇÃO

Agora que as técnicas e ferramentas já foram apresentadas pode-se aplicar o conhecimento obtido na prática. É possível utilizar os algoritmos disponibilizados anteriormente na Seção 6.2 ou então implementar do zero sua aplicação. Veja a seguir o que fazer:

- Implementar um algoritmo de multiplicação de matrizes convencional;
- Separar este algoritmo em funções, por exemplo;
  - Aloca();
  - Inicializa();
  - Multiplica();
  - Desaloca();
- Executar o Gprof e ver qual função é mais custosa;
- Aplicar *Loop Interchange*;
- Analisar com Gprof, Perf e PCM;
- Aplicar *Loop Unrolling*;
- Analisar com Gprof, Perf e PCM;
- Implementar um algoritmo de multiplicação de matrizes com *Loop Tiling*;
- Analisar com Gprof, Perf e PCM;
- Aplicar *Loop Interchange*;
- Analisar com Gprof, Perf e PCM;
- Aplicar *Loop Unrolling*;
- Analisar com Gprof, Perf e PCM;
- Paralelizar;
- Analisar com Gprof, Perf e PCM;

Para saber o tamanho do unroll ou do bloco a ser definido deve-se conhecer o tamanho da memória *cache* da máquina a ser utilizada para aproveitamento total das técnicas. Para isso use o comando ***lscpu***, o qual irá mostrar as configurações do seu processador. Veja na Figura 6.16.

```
sherlon@u02s065908:~$ lscpu
Arquitetura:             x86_64
Modo(s) operacional da CPU:32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  32
On-line CPU(s) list:     0-31
Thread(s) per núcleo  2
Núcleo(s) por soquete:8
Soquete(s):              2
Nó(s) de NUMA:         2
ID de fornecedor:        GenuineIntel
Família da CPU:         6
Modelo:                  45
Model name:              Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
Step:                    7
CPU MHz:                 1312.109
CPU max MHz:             2800,0000
CPU min MHz:             1200,0000
BogoMIPS:                3991.88
Virtualização:         VT-x
cache de L1d:            32K
cache de L1i:            32K
cache de L2:             256K
cache de L3:             20480K
NUMA node0 CPU(s):       0-7,16-23
NUMA node1 CPU(s):       8-15,24-31
```

Figura 6.16. Informações sobre o processador.

O cálculo do bloco a ser definido é baseado no tamanho do nível L1 da memória *cache*, que neste caso é 32KB. Veja o tipo de variável utilizada e quanto ocupa em memória (Use o Algoritmo 6.17), neste caso é *double* e ocupa 8 *bytes* cada elemento, como mostrado na Figura 6.17.

```
1 #include<stdio.h>
2 int main(void){
3   printf("INT:%lu\n", sizeof(int));
4   printf("FLOAT:%lu\n", sizeof(float));
5   printf("DOUBLE:%lu\n", sizeof(double));
6   return 0;
7 }
```

Algoritmo 8: Obter tamanho de tipos de variáveis na memória.

```
shevs@shevs-VirtualBox:~/Área de Trabalho/Fapergs-Unipampa/MINICURSO_ERA
D_2017$ gcc -o programa Tamanho_Variaveis.c
shevs@shevs-VirtualBox:~/Área de Trabalho/Fapergs-Unipampa/MINICURSO_ERA
D_2017$ ./programa
INT: 4
FLOAT: 4
DOUBLE: 8
```

Figura 6.17. Informações sobre as variáveis.

Sabendo que cada elemento da matriz possui 8 *bytes* e que o nível L1 da memória

cache possui 32KB se calcula o número de elementos a serem armazenados, para isso, supondo que será uma matriz de ordem 32 (ou um bloco de ordem 32 de uma matriz maior), faz-se 32*32(Ordem)*2(Matriz A e B)*8(*Double*) = 16384 *bytes*. Convertendo tem-se 16KB, o que é inferior a 32KB de cache L1, sendo assim todos elementos são armazenados em *cache* L1. O mesmo serve para calcular o tamanho do *unroll*.

O ideal é manter um valor inferior ao tamanho da *cache* L1, caso contrario os dados serão armazenados em *cache* L2, sucessivamente. E lembre-se, quanto mais distantes os dados estiverem da CPU, mais lenta será a execução.

6.6. CONSIDERAÇÕES FINAIS

O avanço da pesquisa científica tem aumentado a complexidade da programação e o volume de dados a serem computados. Por causa disso, algoritmos devem ser otimizados e adaptados para aproveitarem toda a potência do *hardware* existente. No entanto, a todo momento surgem novas tendências e o programador deve estar preparado para modificações de código.

Existem muitas técnicas para prover alto desempenho às aplicações. É importante entender tais técnicas e inserí-las em aplicações que demandam alto poder computacional. A utilização de códigos eficientes para uso ótimo da memória *cache* é fundamental para o performance de um programa. Da mesma forma, a execução concorrente em máquinas *multi-core* proporciona ganho de desempenho.

Para entender o fluxo do programa e aplicar as técnicas de otimização é importante usar ferramentas de profile como Gprof, Perf e Intel PCM. Utilizando estas ou outras ferramentas pode-se obter detalhadamente informações a respeito da arquitetura utilizada e do comportamento da aplicação executada.

O conteúdo deste trabalho abordou como começar uma aplicação e analisá-la a fim de buscar trechos de código lentos e otimizados. Além de otimizações, é possível obter o relatório das ferramentas para embasar a tomada de decisão de como as implementações devem ser feitas. Desta forma torna-se possível atingir expressivos ganhos de desempenho em diversas aplicações sequenciais e paralelas.

Agradecimentos

Este trabalho foi realizado com recursos de bolsa PROBIC/FAPERGS 2016 e pela agência de fomento CNPq Edital Universal Processo N. 457684/2014-3.

Referências

[Chaves 2010] Chaves, L. (2010). Dicas para facilitar a criação de programas. Universidade Estadual de Campinas.

[Clapp 2015] Clapp, R. G. (2015). Seismic Processing and the Computer Revolution(s). In *SEG Technical Program Expanded Abstracts 2015*, pages 4832–4837.

[Clapp et al. 2010] Clapp, R. G., Fu, H., and Lindtjorn, O. (2010). Selecting the right hardware for reverse time migration. *The Leading Edge*, 29(1):48–58.

[Intel 2012] Intel (2012). Intel Performance Counter Monitor - A better way to measure CPU utilization.

[Lee et al. 2010] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460.

[Satish et al. 2012] Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., and Dubey, P. (2012). Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society.