Relatório do Laboratório 08 - Organização de Computadores

Grupo:

- Henrique Mateus Teodoro - 23100472
- Rodrigo Schwartz - 23100471

Exercício 1

No exercício 1 pede-se a implementação do seguinte algoritmo, que realiza a soma de uma matriz com outra transposta, ambas com tamanho parametrizável:

```
1  float A[MAX, MAX], B[MAX, MAX]
2  for (i=0; i< MAX; i++) {
3    for (j=0; j< MAX; j++) {
4      A[i,j] = A[i,j] + B[j, i];
5    }
6  }
```

Inicialmente, no segmento .data são declaradas as variáveis que serão utilizadas no decorrer do código para impressão, além de A e B, matrizes que irão armazenar os valores digitados pelo usuário. O tamanho parametrizável da matriz também é definido aqui (MAX).

```
.data
        MAX: .word 4     # É possível alterar este valor para definir o tamanho da matriz
        A: .space 64     # Aloca espaço para a matriz A (matriz 4x4, 4 bytes cada) -> 4x4x4 bytes
        B: .space 64     # Aloca espaço para a matriz B (matriz 4x4, 4 bytes cada) -> 4x4x4 bytes
```

O código inicial, logo após a declaração do segmento .text, cria uma matriz A de tamanho MAX×MAX. O código realiza as seguintes operações:

1. Um contador que será utilizado para preencher a matriz será inicializado em 1.
2. Um loop externo percorre cada linha da matriz.
3. Um loop interno percorre cada coluna da linha atual, calcula o endereço de memória correspondente ao elemento A[i][j] e armazena o valor do contador na matriz (nessa respectiva posição).
4. Os índices de linha, coluna e contador são incrementados conforme necessário até que todos os elementos da matriz sejam armazenados.
5. Assim, a matriz fica no seguinte formato (exemplo utilizando matriz 4x4):

	1	2	3	4
A =	5	6	7	8
	9	10	11	12
	13	14	15	16

```
# ---------------------------- Leitura da Matriz A ----------------------------------------------------
        li $t4, 1
laco_linha_leitura_A:
        li $t1, 0                       # Inicializa j com 0
        bge $t0, $s0, fim_laco_linha_leitura_A   # se (i >= MAX) sai do loop

laco_coluna_leitura_A:
        bge $t1, $s0, fim_laco_coluna_leitura_A  # se (j >= MAX) sai do loop

        mul $t3, $t0, $s0               # t3 = i * MAX
        add $t3, $t1, $t3               # t3 = i * MAX + j
        mul $t3, $t3, 4                 # t3 = (i * MAX + j) * 4
        add $t3, $s1, $t3               # Endereço de A[i][j]

        sw $t4, 0($t3)                  # A[i][j] = t4

        addi $t4, $t4, 1
        addi $t1, $t1, 1

        j laco_coluna_leitura_A

fim_laco_coluna_leitura_A:

        addi $t0, $t0, 1

        j laco_linha_leitura_A

fim_laco_linha_leitura_A:
```

Essa mesma estrutura de código é reaproveitada para escrever os valores da matriz B, mudando somente alguns endereços e registradores. Assim, as matrizes A e B, são preenchidas com o contador, tendo um tamanho parametrizável.

Depois disso, a soma de matrizes é realizada. O código soma os elementos da matriz A com a transposta de B e armazena o resultado na matriz A. O código utiliza dois loops aninhados para percorrer todas as posições das matrizes. O código segue a seguinte lógica de execução:

1. Inicializa os contadores de linha (i) e coluna (j) com 0.
2. O loop externo (laco_linha) percorre cada linha da matriz.
3. O loop interno (laco_coluna) percorre cada coluna da matriz.
4. Para cada posição (i,j):
    - Calcula o endereço do elemento A[i][j].
    - Calcula o endereço do elemento B[i][j].
    - Lê os valores A[i][j] e B[i][j].
    - Soma esses valores e armazena o resultado em A[i][j].
5. Os loops incrementam os contadores e continuam até percorrer todos os elementos das matrizes.

```
# ---------------------------- Cálculo da soma ------------------------------------------------
        li $t0, 0               # contador do primeiro loop (i)
        li $t1, 0               # contador do segundo loop (j)

laco_linha:
        li $t1, 0                       # Inicializa j com 0
        bge $t0, $s0, fim_laco_linha    # se (i >= MAX) sai do loop

laco_coluna:
        bge $t1, $s0, fim_laco_coluna   # se (j >= MAX) sai do loop

        mul $t3, $t0, $s0               # t3 = i * MAX
        add $t3, $t1, $t3               # t3 = i * MAX + j
        mul $t3, $t3, 4                 # t3 = (i * MAX + j) * 4
        add $t3, $s1, $t3               # Endereço de A[i][j]

        mul $t4, $t1, $s0               # t4 = j * MAX
        add $t4, $t0, $t4               # t4 = j * MAX + i
        mul $t4, $t4, 4                 # t4 = (j * MAX + i) * 4
        add $t4, $s2, $t4               # Endereço de B[i][j]

        lw $t5, 0($t3)                  # t5 = A[i][j]
        lw $t6, 0($t4)                  # t6 = B[i][j]
        add $t5, $t5, $t6               # t5 = A[i][j] + B[i][j]


        sw $t5, 0($t3)                   # A[i][j] = t5

        addi $t1, $t1, 1

        j laco_coluna

fim_laco_coluna:

        addi $t0, $t0, 1
        j laco_linha

fim_laco_linha:
        nop
```

Exercício 2

O exercício 2 consiste na implementação do seguinte algoritmo:

```
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
  for (j=0; j< MAX; j+=block_size) {
    for (ii=i; ii<i+block_size; ii++) {
      for (jj=j; jj<j+block_size; jj++) {
        A[ii,jj] = A[ii,jj] + B[jj, ii];
      }
    }
  }
}
```

Este algoritmo basicamente é utilizado para realizar a soma de uma matriz A com outra matriz B transposta, assim como no exercício 1. No entanto, esse algoritmo se beneficia da técnica de cache blockin, visando aumentar o Cache Hit Rate.

No segmento .data, são declaradas as variáveis que serão úteis ao longo do programa:

```
.data
        MAX: .word 4    # É possível alterar este valor para definir o tamanho da matriz
        block_size: .word 2     # É possível alterar este valor para definir o tamanho do bloco da matriz
        A: .space 64    # Aloca espaço para a matriz A (matriz 4x4, 4 bytes cada) -> 4x4x4 bytes
        B: .space 64    # Aloca espaço para a matriz B (matriz 4x4, 4 bytes cada) -> 4x4x4 bytes
```

Há a declaração das matrizes A e B, do tamanho das matrizes, e uma novidade, que consiste na variável block_size, que indica o tamanho do bloco que será pego na utilização da técnica do cache blocking. Além de variáveis utilizadas para impressão no console.

Depois disso, as matrizes A e B são preenchidas, assim como no exercício 1:

```
# ---------------------------- Leitura da Matriz A ----------------------------------------------------
        li $t4, 1
laco_linha_leitura_A:
        li $t1, 0                       # Inicializa j com 0
        bge $t0, $s0, fim_laco_linha_leitura_A   # se (i >= MAX) sai do loop

laco_coluna_leitura_A:
        bge $t1, $s0, fim_laco_coluna_leitura_A  # se (j >= MAX) sai do loop

        mul $t3, $t0, $s0               # t3 = i * MAX
        add $t3, $t1, $t3               # t3 = i * MAX + j
        mul $t3, $t3, 4                 # t3 = (i * MAX + j) * 4
        add $t3, $s1, $t3               # Endereço de A[i][j]

        sw $t4, 0($t3)                  # A[i][j] = t4

        addi $t4, $t4, 1
        addi $t1, $t1, 1

        j laco_coluna_leitura_A

fim_laco_coluna_leitura_A:

        addi $t0, $t0, 1

        j laco_linha_leitura_A

fim_laco_linha_leitura_A:
```

Após as matrizes A e B terem seus valores preenchidos adequadamente, parte-se para a parte de cálculo e computação. Onde serão somados os valores da matriz A, com os valores da matriz B transposta, entrada a entrada.

Para realizar essa computação, foram utilizados 4 laços for, seguindo corretamente o algoritmo destacado acima:

```
# ---------------------------- Cálculo da soma ----------------------------------------------
        li $t0, 0               # contador do primeiro loop (i)
        li $t1, 0               # contador do segundo loop (j)
        li $t2, 0               # contador do terceiro loop (ii)
        li $t3, 0               # contador do quarto loop (jj)
        li $t4, 0               # comparador i+block_size
        li $t5, 0               # comparador j+block_size

laco_i:
        li $t1, 0               # seta j para 0
        beq $t0, $s0, fim_laco_i  # se i >= MAX sai do loop

laco_j:
        move $t2, $t0           # faz ii = i

        beq $t1, $s0, fim_laco_j  # se j >= MAX sai do loop

laco_ii:
        move $t3, $t1           # faz jj = j
        add $t4, $t0, $s3       # faz i + block_size

        beq $t2, $t4, fim_laco_ii  # se ii >= i+block_size sai do loop
```

```
laco_jj:
        add $t5, $t1, $s3       # faz j + block_size

        beq $t3, $t5, fim_laco_jj  # se jj >= j+block_size sai do loop

        mul $t6, $t2, $s0       # $t6 = ii * MAX
        add $t6, $t6, $t3       # $t6 = ii * MAX + jj
        mul $t6, $t6, 4         # $t6 = (ii * MAX + jj) * 4
        add $t6, $t6, $s1       # Endereço de A[ii][jj]

        mul $t7, $t3, $s0       # $t7 = jj * MAX
        add $t7, $t7, $t2       # $t7 = jj * MAX + ii
        mul $t7, $t7, 4         # $t7 = (jj * MAX + ii) * 4
        add $t7, $t7, $s2       # Endereço de B[jj][ii]

        lw $t8, 0($t6)          # $t8 = A[ii][jj]
        lw $t9, 0($t7)          # $t9 = B[ii][jj]
        add $s4, $t8, $t9       # $s4 = A[ii][jj] + B[ii][jj]
        sw $s4, 0($t6)          # A[ii][jj] = A[ii][jj] + B[ii][jj]

        addi $t3, $t3, 1        # faz jj++
        j laco_jj               # retorna para o laço jj

fim_laco_jj:
        addi $t2, $t2, 1        # faz ii++
        j laco_ii               # retorna para o laço ii

fim_laco_ii:
        add $t1, $t1, $s3       # faz j += block_size
        j laco_j                # retorna para o laço j

fim_laco_j:
        add $t0, $t0, $s3       # i += block_size
        j laco_i                # retorna para o laço i

fim_laco_i:
        nop
```

Ao término dos cálculos, o valor da soma de A com a transposta de B é armazenado na matriz A.

Exercício 3

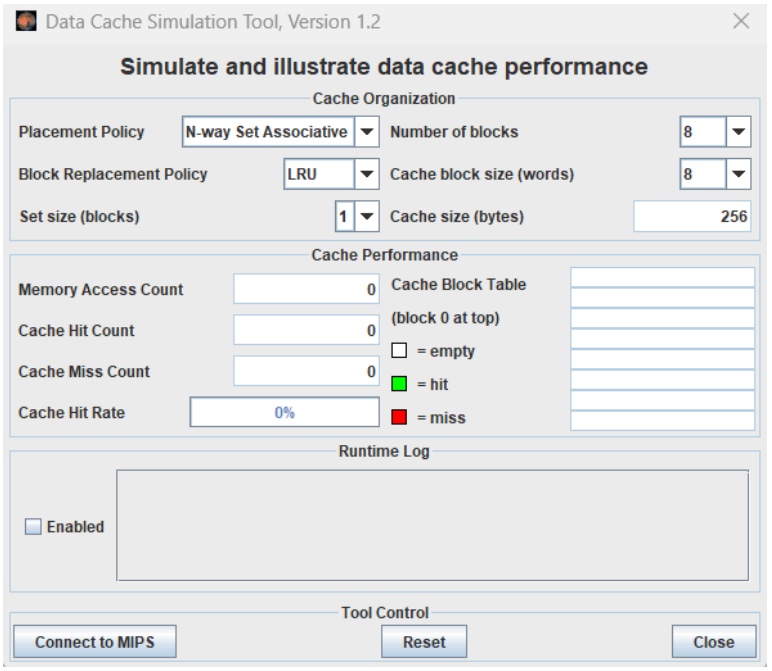
A seguir segue o resultado dos testes utilizando a Data Cache Simulator variando alguns parâmetros (organizamos os dados em tabelas para facilitar a leitura:
*OBS: O placement policy utilizado foi N-way Set Associative

Primeiro teste:

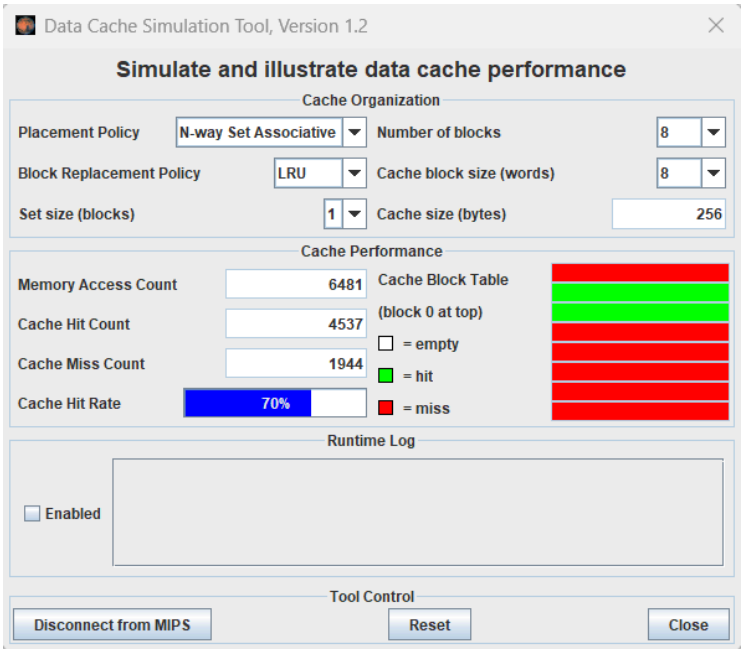
Vamos realizar o teste dos dois algoritmos, visando perceber se há um ganho significativo e considerável utilizando-se blocking cache. Para fins de teste, as matrizes A e B terão tamanho de 36 x 36 (36 linhas por 36 colunas), cujos

elementos de ambas estarão em ordem crescente, partindo do valor 1. Ou seja, a matriz A terá em sua 1 linha os valores 1, 2, …, 36, na sua segunda linha os valores 37, 38, .., 72, e assim, sucessivamente. A matriz B será preenchida de forma idêntica. Após as matrizes serem preenchidas, serão realizados os cálculos, onde a matriz A será somada com a matriz B transposta, entrada a entrada. Vamos analisar os resultados utilizando o primeiro algoritmo (exercício 1) que não utiliza blocking cache, e os resultados para o segundo algoritmo (exercício 2), que utiliza a técnica de blocking cache.

Obs: Para ambos os testes utilizou-se uma cache 8x8, ou seja, uma cache que possui 8 blocos e 8 words por bloco. Além disso, a forma de mapeamento para cache escolhida foi o N-way Set Associative:

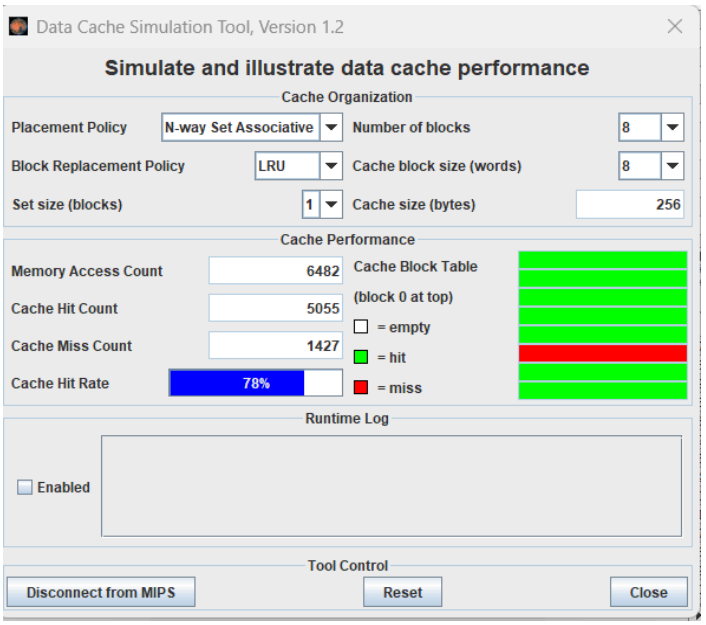


Para o algoritmo do exercício 1 obteve-se o seguinte resultado:



Houve um Cache Hit Rate de 70%, e consequentemente, houve um Cache Miss Rate de 30%. A maior parte dos misses ocorrem devido a matriz B, visto que como a soma deve ser feita entre a matriz A e a matriz B transposta, quando são buscados os valores da matriz B transporta, o bloco buscado não trará juntamente os elementos que serão buscados para serem somados nas próximas iterações. Exemplificando melhor isso, quando o valor 1 da matriz B transposta for buscado, serão trazidos no mesmo bloco os valores 2, 3, …, 36. No entanto, os valores buscados em ordem pela transposta serão: 1, 37, 73, …, etc. Com isso, serão gerados muitos misses ao longo da execução, visto que não se está tirando proveito do princípio da localidade espacial.

Para o algoritmo do exercício 2, obteve-se o seguinte resultado, utilizando-se um block size de 6:



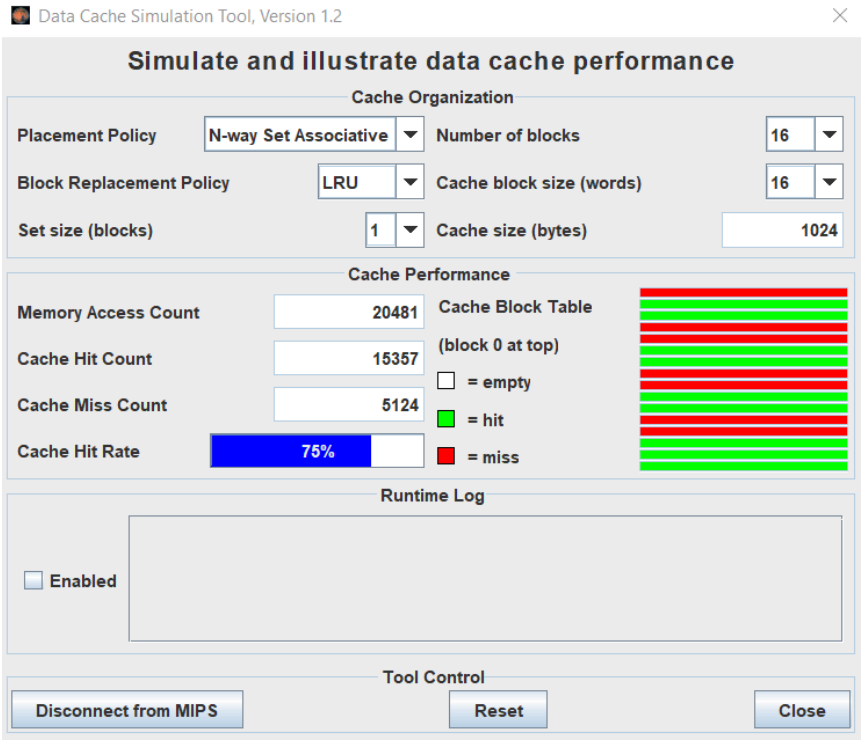
Houve um Cache Hit Rate de 78%, e consequentemente, um Cache Miss Rate de 22%. Percebe-se que a utilização do método de blocking cache, ou seja, a divisão das matrizes em blocos menores (tamanho dos blocos sendo 6), fez com que houvesse um aproveitamento um pouco melhor da localidade espacial, possibilitando que haja uma redução no número de misses, visto que estão sendo colocados valores mais próximos na cache, possibilitando um acesso mais eficiente.

Ambos os casos apresentaram resultados intermediários, visto que o tamanho da cache é consideravelmente pequeno (8x8), em comparação com as matrizes utilizadas (36x36), acarretando em uma quantidade significativa de misses.

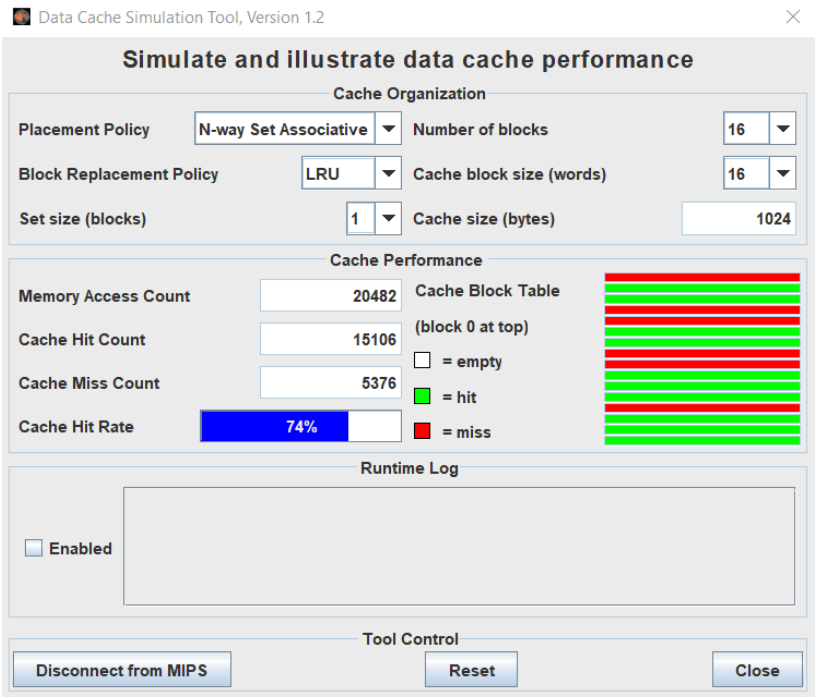
Segundo teste:

Testamos ambos os algoritmos para ver se o cache de blocos oferece um ganho significativo. Para fins de teste, as matrizes A e B terão tamanho de 64x64, e seus elementos serão classificados em ordem crescente, começando com o valor 1 (a matriz será preenchida sequencialmente pela linha por um contador, assim cada posição vai sendo preenchida com 1, 2, 3, 4, 5, 6 … até chegar em seu fim). A Matriz B será preenchida da mesma maneira. Os cálculos começam após completar as matrizes. Neste processo, será realizada a soma da matriz A com a transposta da matriz. O resultado é armazenado na matriz A. Vamos analisar os resultados do exercício 1, que não utiliza blocking cache, e os resultados do exercício 2, que utiliza a técnica de blocking cache. A forma de mapeamento para cache escolhida foi o N-way Set Associative, o number of blocks será de 16 e o cache block size (words) será de 16 também, detalhar um desempenho melhor entre as soluções.

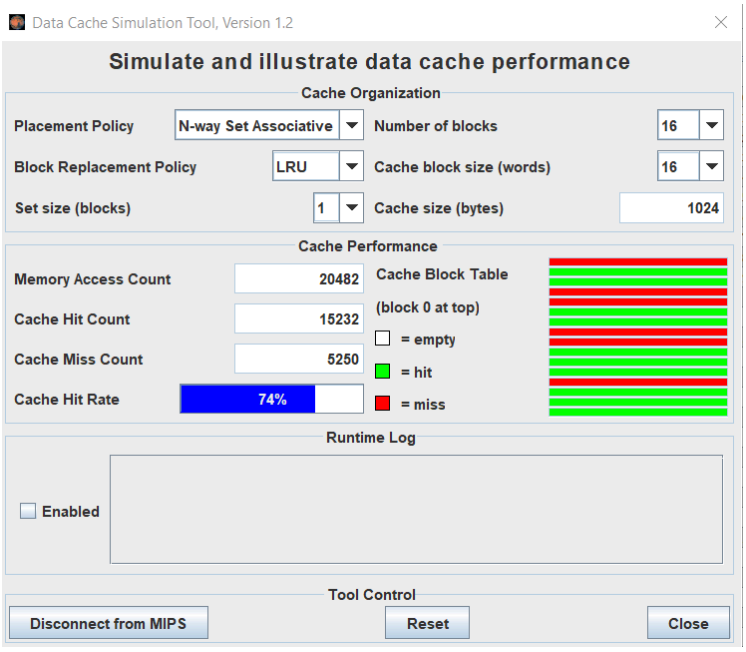
Para o exercício 1 obtivemos o seguinte resultado:



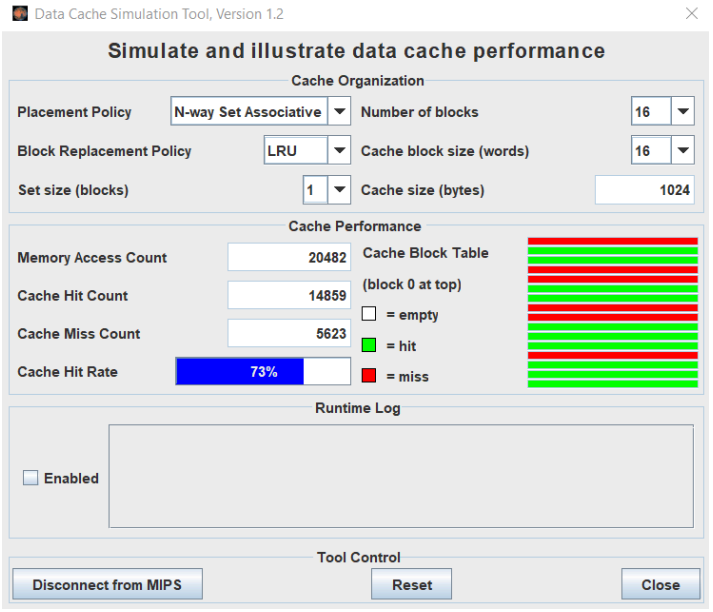
Houve um Cache Hit Rate de 75% e um Cache Miss Rate de 25%. A maior parte dos misses, como já explicado anteriormente, ocorrem por estarmos buscando a matriz transposta de B. Assim, o bloco buscado não trará juntamente os elementos que serão buscados para serem somados nas próximas iterações, ocorrendo misses mais frequentemente. Para o exercício 2, com block_size igual a 16, obtivemos o seguinte resultado:



Houve um Cache Hit Rate de 74% e um Cache Miss Rate de 26%. Para o exercício 2, com block_size igual a 32, obtivemos o seguinte resultado:



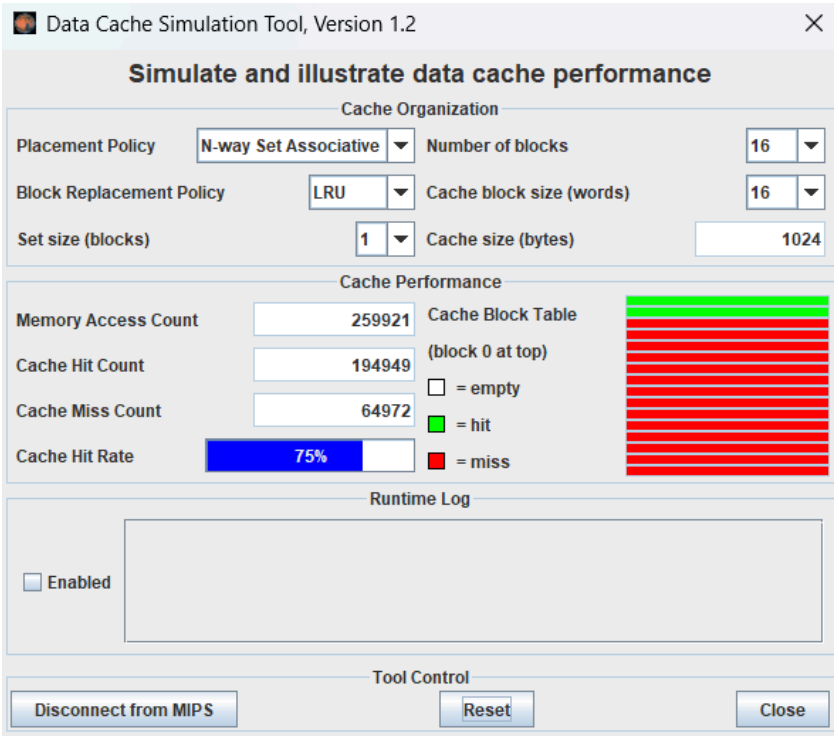
Houve um Cache Hit Rate de 74% e um Cache Miss Rate de 26%. Para o exercício 2, com block_size igual a 8, obtivemos o seguinte resultado:



Houve um Cache Hit Rate de 73% e um Cache Miss Rate de 27%. Nota-se que, mesmo utilizando a técnica de blocking size, ainda há casos em que sua utilização não é a mais recomendada. Notamos, ao decorrer dos testes que fizemos para o laboratório 08, que nos casos onde o tamanho da matriz é uma potência de 2, a utilização da técnica de blocking cache é indiferente ou até piora o desempenho. O teste 2, realizado agora, é um exemplo disso, como 64 é uma potência de 2 (2^8 = 64), a utilização de blocking cache não mostrou nenhum ganho de desempenho, e somente piorou em alguns casos. Acreditamos que quando tamanho da matriz é uma potência de 2, conflitos de cache que ocorrem com maior frequência e ocorrem mais misses, aumentando, consequentemente, a Cache Miss Rate.

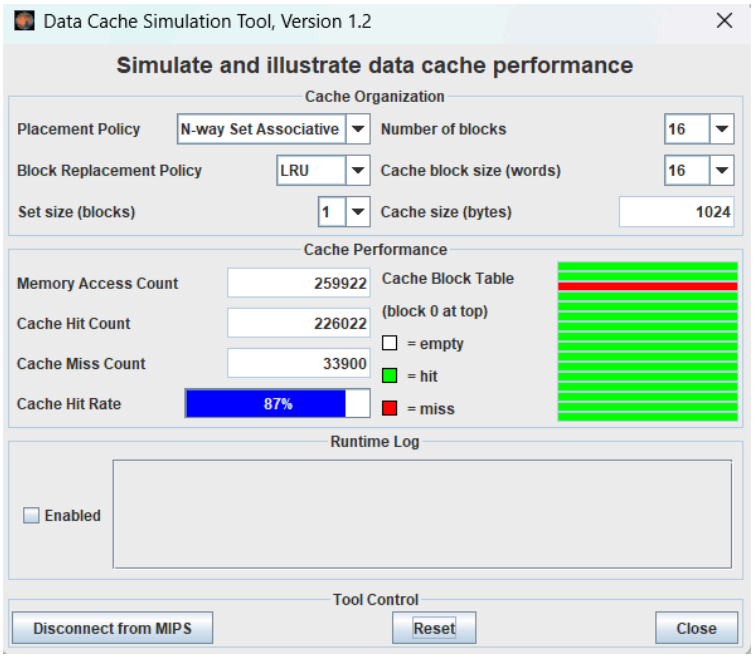
Terceiro teste:

Vamos analisar o resultado de ambos os algoritmos para matrizes maiores. Para este teste, utilizamos matrizes de tamanho 216 x 216, em uma memória cache de tamanho 16 x 16, com o mapeamento N-way Set Associative:
Para o primeiro algoritmo, obteve-se o seguinte resultado:



Houve um Cache Hit Rate de 75% e um Cache Miss Rate de 25%.

Para o segundo algoritmo, obteve-se o seguinte resultado:



Houve um Cache Hit Rate de 87% e um Cache Miss Rate de 13%.

Com os resultados nos testes apresentados, é possível perceber que existem alguns padrões e parâmetros que potencializam o ganho utilizando o método de blocking cache. O tamanho das matrizes é algo que deve ser considerado, visto que

matrizes maiores tendem a aproveitar melhor a técnica de blocking cache. Isso ocorre porque, se as matrizes forem muito pequenas, não haverá um ganho considerável em se dividir as matrizes, e processá-las em blocos menores, visto que as matrizes já são de fato pequenas. Para matrizes maiores, há um melhor aproveitamento ao se dividir as matrizes, aproveitando mais a localidade espacial e com isso, não será necessário acessar tantas vezes a memória principal, diminuindo assim os misses.

Outra coisa importante é o tamanho do bloco que será utilizado na técnica de blocking cache, e claro, o tamanho da própria memória cache. O tamanho do bloco deve ser capaz de caber na memória cache. Quanto mais dados de um bloco couberem na cache melhor, porque com isso, o processamento desses dados nos cálculos serão feitos de modo a minimizar os misses, visto que não será necessário sair até a memória principal para buscar um dado faltante. Com isso, se o bloco for muito maior que a cache, os dados do bloco inteiro não irão caber na cache, e com isso, haverá desperdício de desempenho, devido ao não aproveitamento do princípio da localidade espacial.

Por fim, o tamanho da cache também importa. Caches grandes propiciam um ganho de desempenho, visto que permitem uma maior quantidade de dados dentro dela. Assim, o blocking cache é beneficiado, visto que o tamanho do bloco (block size) pode ser aumentado, utilizando mais o princípio da localidade espacial, e assim, gerando mais hits.

Com isso, conclui-se que, com uma utilização adequada, é possível obter-se um ganho significativo de desempenho através da técnica de blocking cache, em situações de cálculo envolvendo matrizes, por exemplo.