Thực hành kiến trúc máy tính

Báo cáo thực hành

Bài 12. Bộ nhớ đệm nhanh – Cache memory

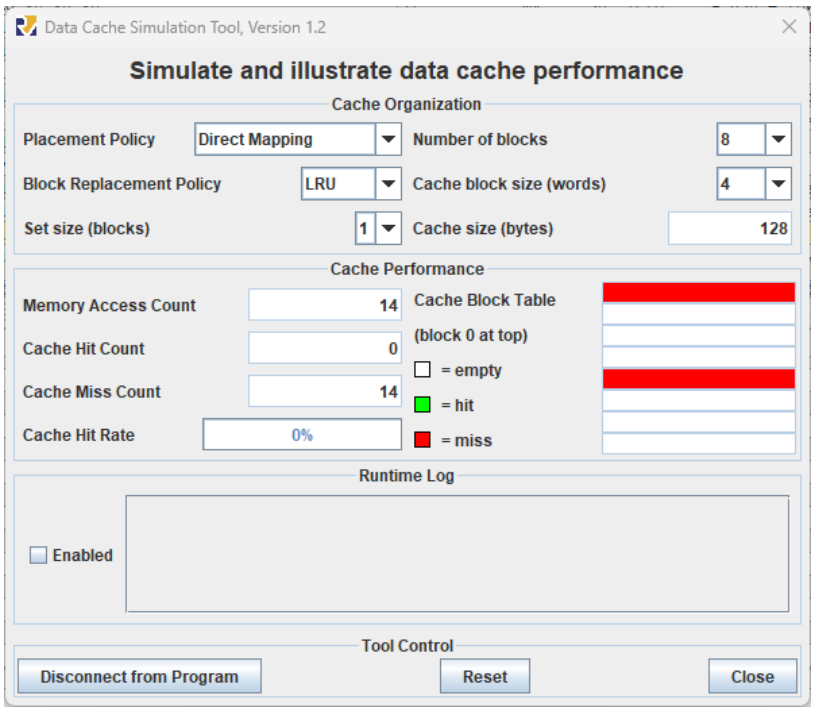
Họ Tên	Lê Thành An
MSSV	20235631

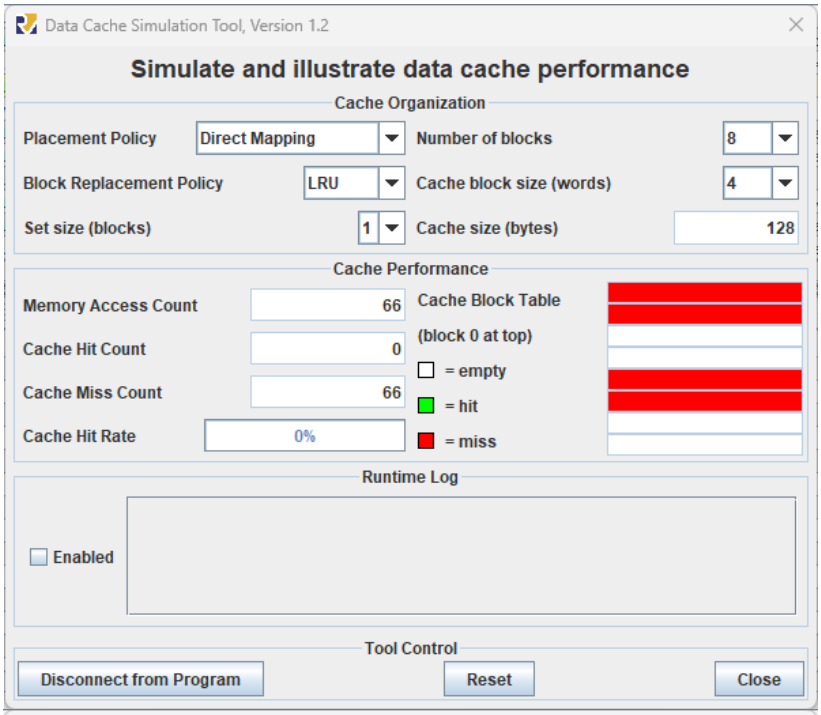
ASSIGNMENT 1

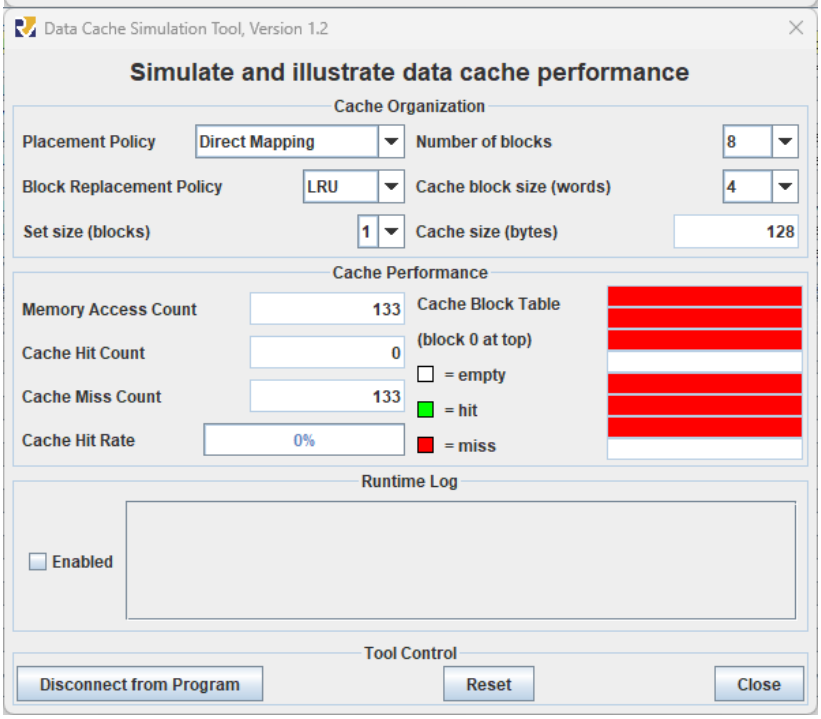
ĐOẠN MÃ :

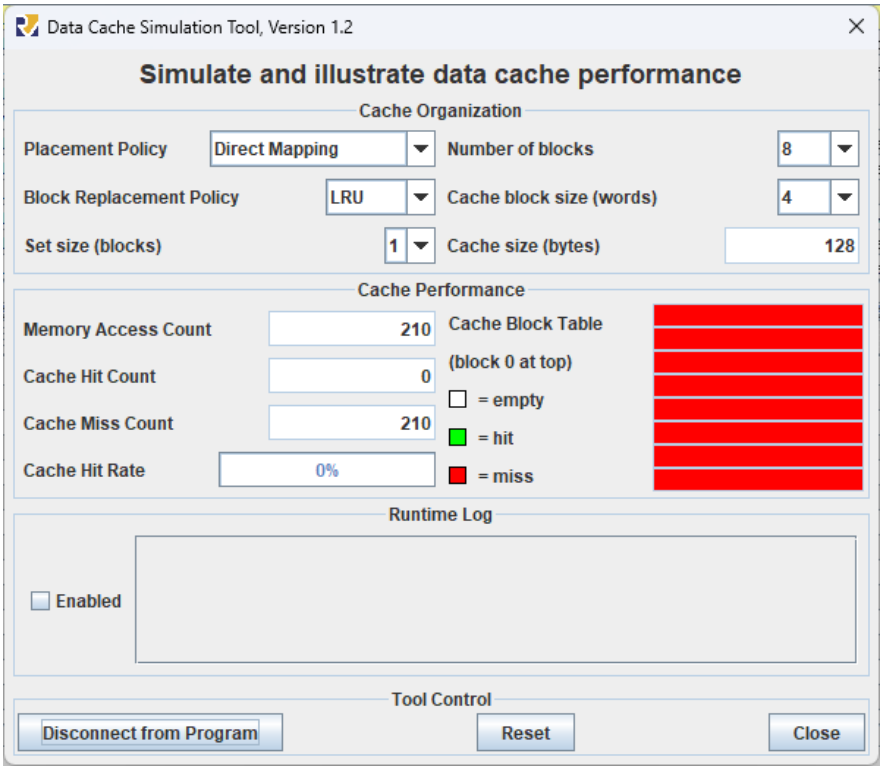
```
#################################################################
#
#  Column-major order traversal of 16 x 16 array of words.
#  Pete Sanderson
#  31 March 2007
#
#  To easily observe the column-oriented order, run the Memory Reference
#  Visualization tool with its default settings over this program.
#  You may, at the same time or separately, run the Data Cache Simulator
#  over this program to observe caching performance.  Compare the results
#  with those of the row-major order traversal algorithm.
#
#  The C/C++/Java-like equivalent of this MIPS program is:
#     int size = 16;
#     int[size][size] data;
#     int value = 0;
#     for (int col = 0; col < size; col++) {
#        for (int row = 0; row < size; row++) }
#           data[row][col] = value;
#           value++;
#        }
#     }
#
#  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change
this,
#        three statements need to be changed.
#        1. The array storage size declaration at "data:" needs to be
changed from
#           256 (which is 16 * 16) to #columns * #rows.
#        2. The "li" to initialize $t0 needs to be changed to the new
#rows.
#        3. The "li" to initialize $t1 needs to be changed to the new
#columns.
#
         .data
data:    .word     0 : 256       # 16x16 matrix of words
```

```
        .text
        li       t0, 16         # $t0 = number of rows
        li       t1, 16         # $t1 = number of columns
        la        a0, data
        mv     s0, zero      # $s0 = row counter
        mv     s1, zero      # $s1 = column counter
        mv     t2, zero      # $t2 = the value to be stored
#  Each loop iteration will store incremented $t1 value into next element
of matrix.
#  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
#  Note: no attempt is made to optimize runtime performance!
loop:    mul     s2, s0, t1        # $s2 = row * #cols  (two-instruction
sequence)
        add      s2, s2, s1   # $s2 += col counter
        slli      s2, s2, 2     # $s2 *= 4 (shift left 2 bits) for byte
offset
        add       s2, a0, s2
        sw       t2, 0(s2) # store the value in matrix element
        addi     t2, t2, 1     # increment value to be stored
#  Loop control: If we increment past bottom of column, reset row and
increment column
#               If we increment past the last column, we're finished.
        addi     s0, s0, 1     # increment row counter
        bne      s0, t0, loop # not at bottom of column so loop back
        mv     s0, zero      # reset row counter
        addi     s1, s1, 1     # increment column counter
        bne      s1, t1, loop # loop back if not at end of matrix (past
the last column)
#  We're finished traversing the matrix.
        li       a7, 10         # system service 10 is exit
        ecall                  # we are outta here.
```




Data Cache Simulation Tool, Version 1.2
Simulate and illustrate data cache performance
Cache Organization
Placement Policy
Direct Mapping
Number of blocks
8
Block Replacement Policy
LRU
Cache block size (words)
4
Set size (blocks)
1
Cache size (bytes)
128
Cache Performance
Memory Access Count
66
Cache Hit Count
0
Cache Miss Count
66
Cache Hit Rate
0%
Cache Block Table
(block 0 at top)
= empty
= hit
= miss
Runtime Log
Enabled
Tool Control
Disconnect from Program
Reset
Close


Data Cache Simulation Tool, Version 1.2
Simulate and illustrate data cache performance
Cache Organization
Placement Policy
Direct Mapping
Number of blocks
8
Block Replacement Policy
LRU
Cache block size (words)
4
Set size (blocks)
1
Cache size (bytes)
128
Cache Performance
Memory Access Count
133
Cache Hit Count
0
Cache Miss Count
133
Cache Hit Rate
0%
Cache Block Table
(block 0 at top)
= empty
= hit
= miss
Runtime Log
Enabled
Tool Control
Disconnect from Program
Reset
Close



1. Ảnh hưởng của Block Size (Kích thước khối):

- Block size nhỏ:
  - Khi một phần tử được truy cập và không có trong cache (cache miss), một khối dữ liệu xung quanh phần tử đó sẽ được nạp vào cache. Nếu kích thước khối nhỏ, chỉ một vài phần tử lân cận (theo địa chỉ bộ nhớ) được đưa vào cache.
  - Trong chương trình duyệt theo cột này, các phần tử được truy cập liên tiếp nằm cách xa nhau trong bộ nhớ (64 bytes). Do đó, khi một khối nhỏ được nạp, các lần truy cập tiếp theo trong cùng cột rất có khả năng gây ra cache miss, vì các phần tử cần thiết không nằm trong khối vừa được nạp.
  - Kết quả: Tỷ lệ cache hit thấp.
- Block size lớn:
  - Nếu kích thước khối đủ lớn (lớn hơn hoặc bằng khoảng cách giữa các phần tử liên tiếp trong một cột là 64 bytes), khi một phần tử được truy cập lần đầu tiên, khối chứa phần tử đó có thể bao gồm cả các phần tử tiếp theo trong cùng cột.
  - Tuy nhiên, vì chúng ta đang duyệt theo cột, các phần tử liên tiếp được truy cập sẽ thuộc các khối khác nhau trong bộ nhớ. Một khối lớn có thể chứa nhiều phần tử của các hàng khác nhau, nhưng khi chuyển sang cột tiếp theo, những phần tử này có thể không được sử dụng lại ngay.
  - Kết quả: Tỷ lệ cache hit có thể cải thiện so với block size nhỏ, nhưng vẫn có thể không tối ưu do đặc điểm truy cập theo cột.

2. Ảnh hưởng của Number of Blocks (Số lượng khối):

- Số lượng khối nhỏ:
  - Cache có ít chỗ để lưu trữ các khối dữ liệu. Khi chương trình truy cập các khối dữ liệu mới, các khối cũ sẽ dễ dàng bị loại bỏ (do các chính sách thay thế cache như LRU - Least Recently Used).
  - Trong trường hợp duyệt theo cột, khi chuyển từ một cột sang cột tiếp theo, các khối dữ liệu của cột trước đó có thể đã bị loại bỏ, dẫn đến nhiều cache miss hơn khi truy cập các phần tử của cột

mới lần đầu tiên.
    - Kết quả: Tỷ lệ cache hit thấp.
- Số lượng khối lớn:
    - Cache có nhiều chỗ hơn để lưu trữ các khối dữ liệu khác nhau. Điều này giúp giảm số lần các khối cần thiết bị loại bỏ và sau đó phải nạp lại.
    - Tuy nhiên, với cách truy cập theo cột, việc tăng số lượng khối có thể không mang lại lợi ích đáng kể sau một ngưỡng nhất định. Lý do là vì các lần truy cập liên tiếp trong thời gian ngắn thường cách xa nhau trong bộ nhớ, và một cache lớn cũng không thể lưu trữ toàn bộ mảng cùng một lúc (256 words * 4 bytes/word = 1024 bytes).
    - Kết quả: Tỷ lệ cache hit có thể cao hơn so với số lượng khối nhỏ, nhưng hiệu quả bị giới hạn bởi pattern truy cập.

Tóm lại:

Đối với chương trình duyệt mảng theo thứ tự cột này:

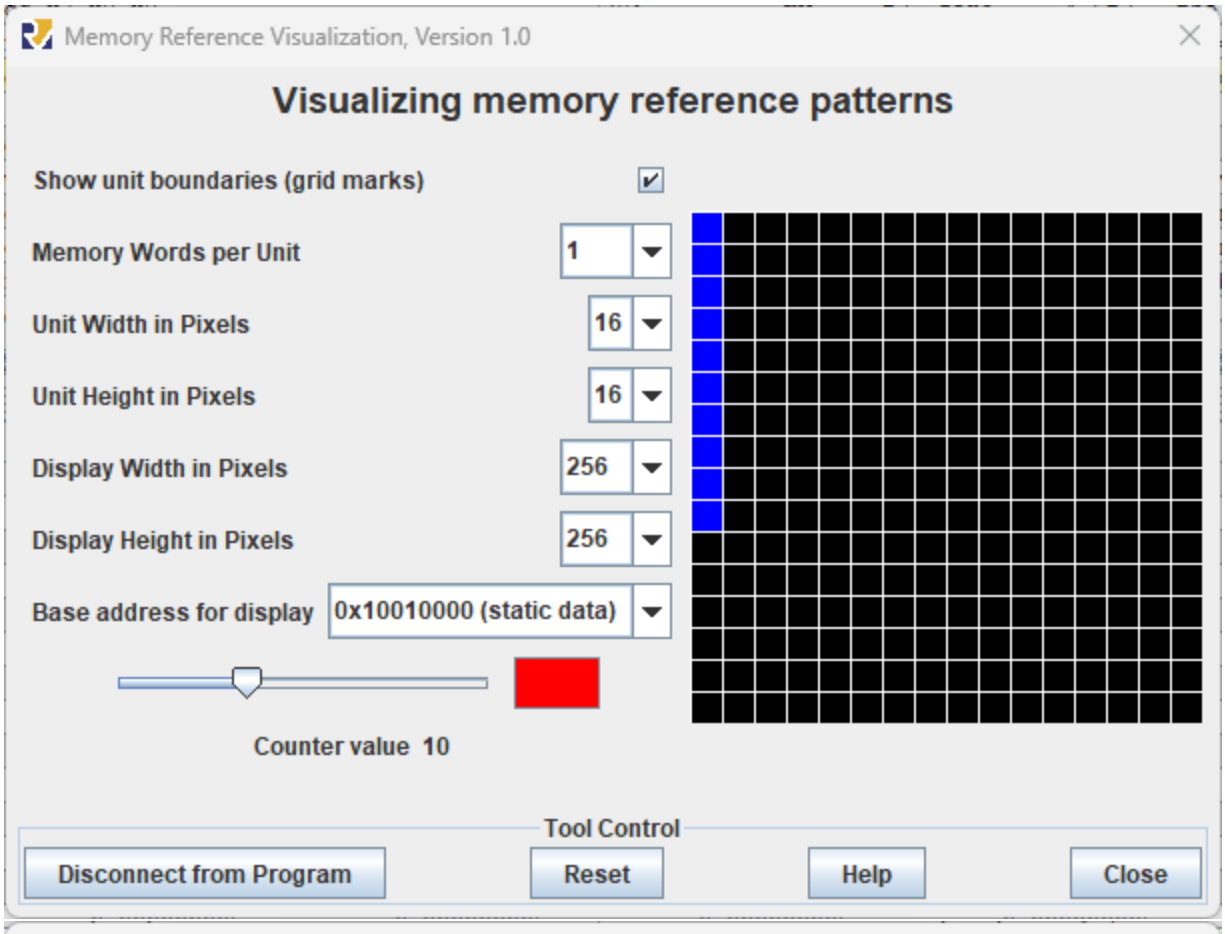
- Block size: Một block size quá nhỏ sẽ dẫn đến tỷ lệ cache hit rất thấp do tính chất truy cập cách quãng theo cột. Một block size lớn hơn có thể cải thiện, nhưng không tối ưu vì không tận dụng được tính cục bộ không gian theo hàng.
- Number of Blocks: Số lượng khối càng lớn thì khả năng lưu trữ nhiều khối khác nhau càng cao, giúp giảm số lần cache miss do xung đột dung lượng. Tuy nhiên, hiệu quả của việc tăng số lượng khối bị giới hạn bởi pattern truy cập theo cột, nơi các lần truy cập gần nhau về thời gian lại xa nhau về không gian bộ nhớ.

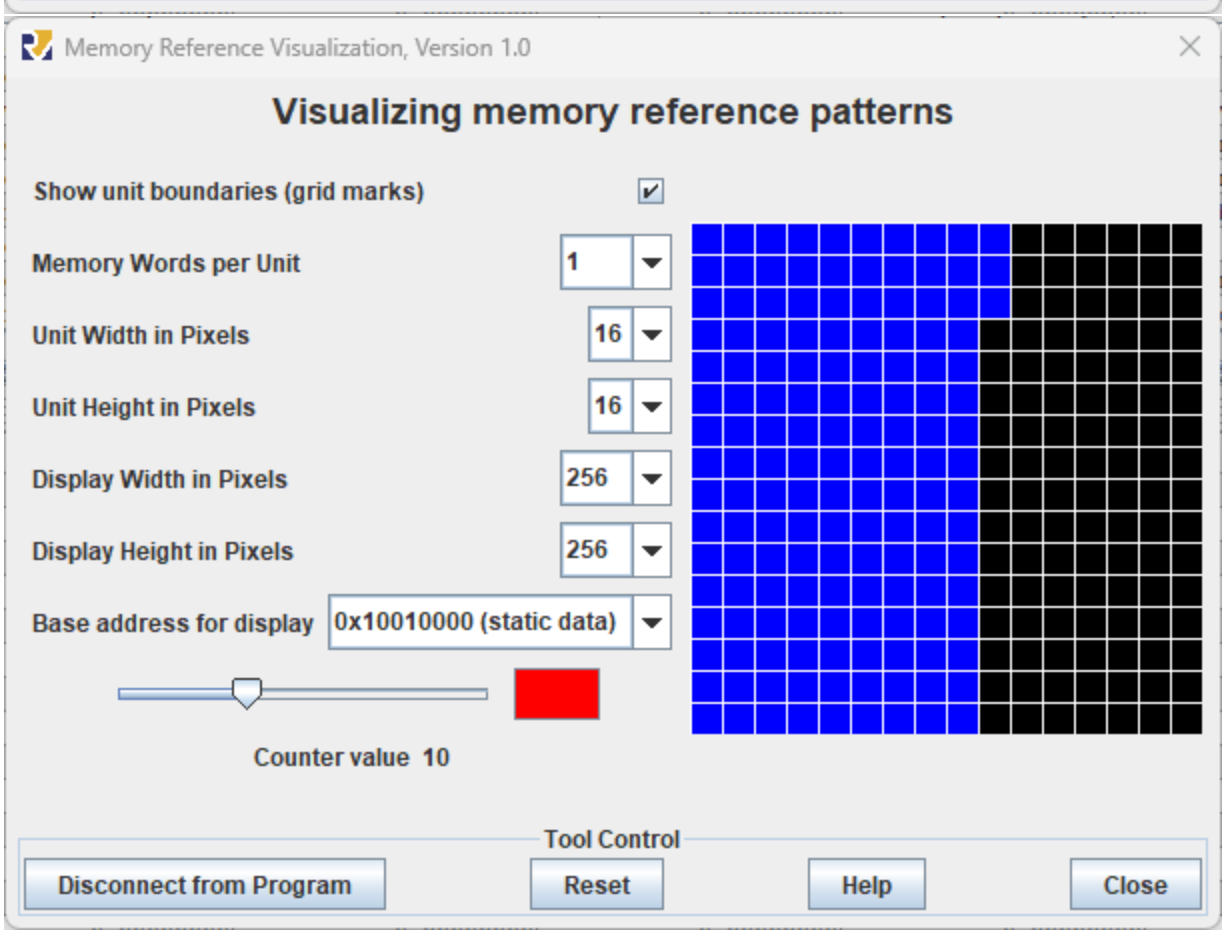
So sánh với duyệt theo hàng (Row-major order):

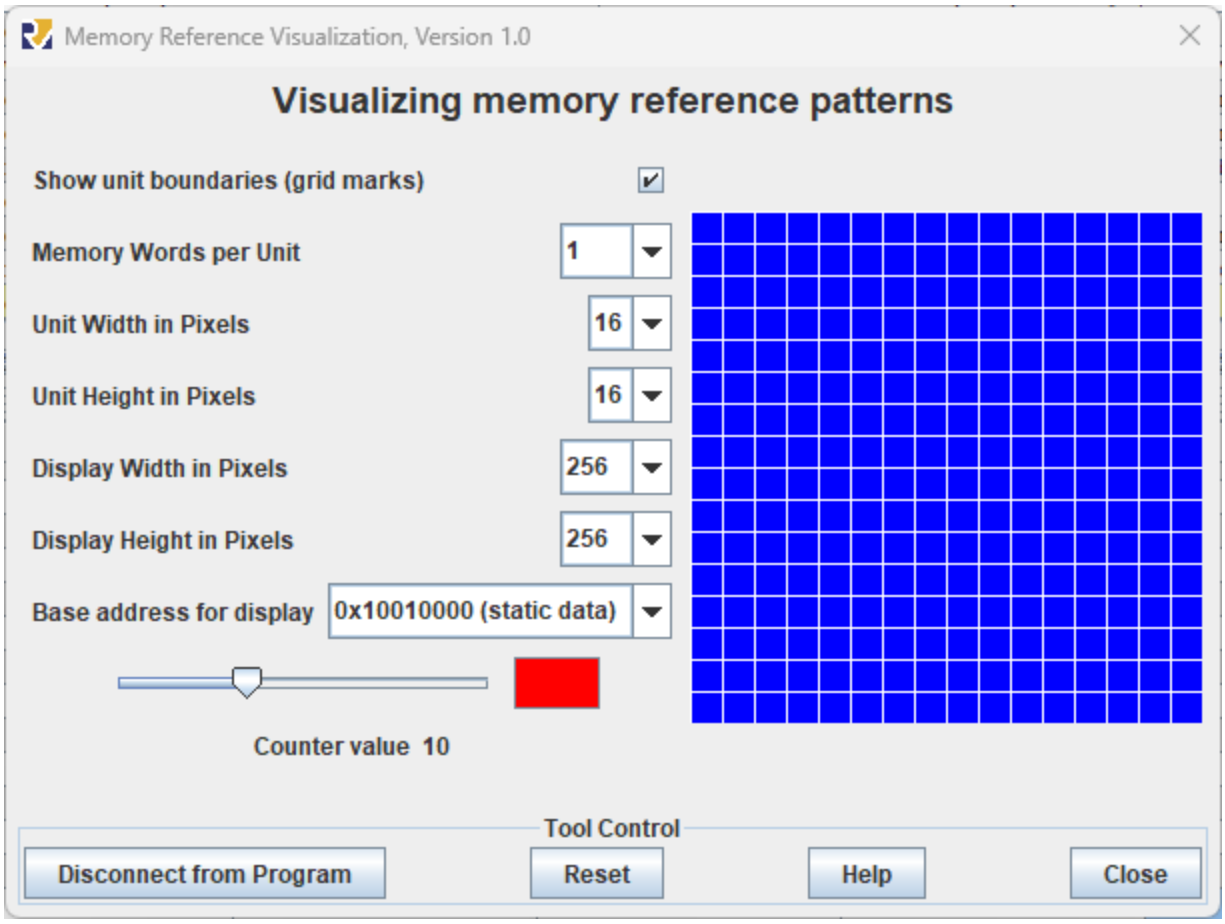
Nếu chương trình duyệt mảng theo thứ tự hàng, các phần tử được truy cập liên tiếp sẽ nằm liền kề nhau trong bộ nhớ. Trong trường hợp đó:

- Block size: Một block size phù hợp có thể chứa nhiều phần tử liên tiếp trong một hàng. Khi một khối được nạp, các lần truy cập tiếp theo vào các phần tử kế cận sẽ là cache hit, dẫn đến tỷ lệ cache hit cao hơn nhiều.
- Number of Blocks: Số lượng khối lớn vẫn hữu ích để lưu trữ nhiều hàng đã được truy cập gần đây, giảm cache miss khi chuyển sang các hàng tiếp theo.

Kết quả:


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close



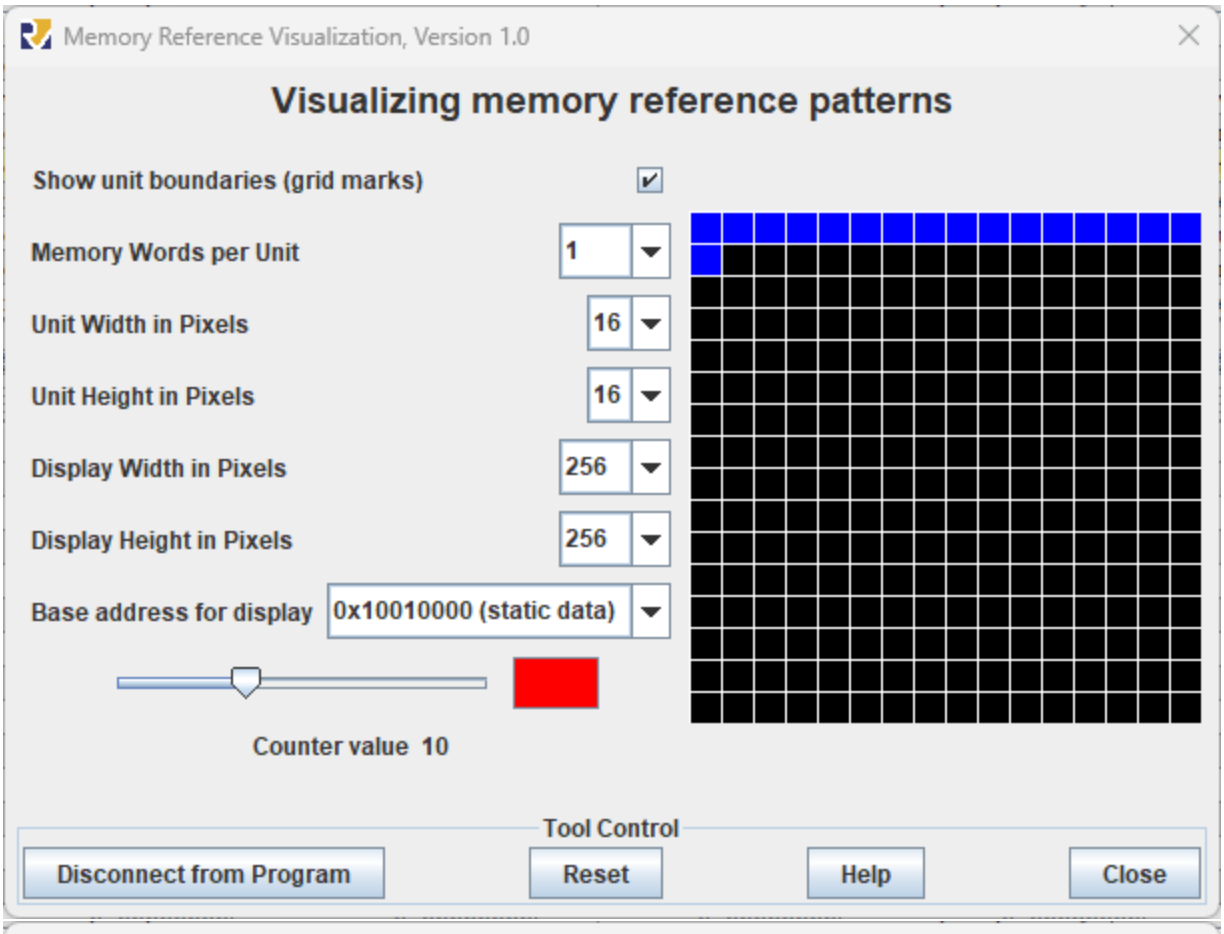
ASSIGNMENT 2

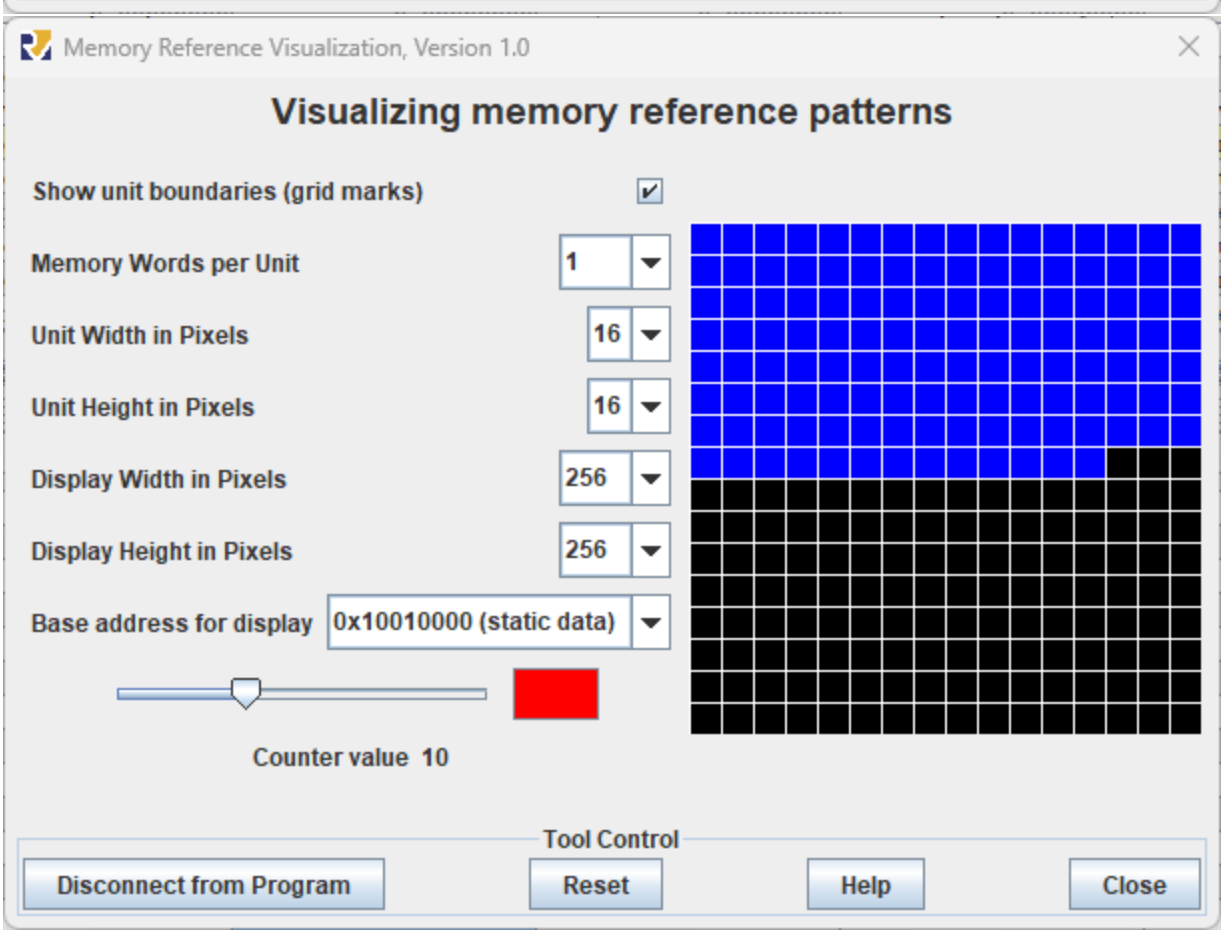
ĐOẠN MÃ :

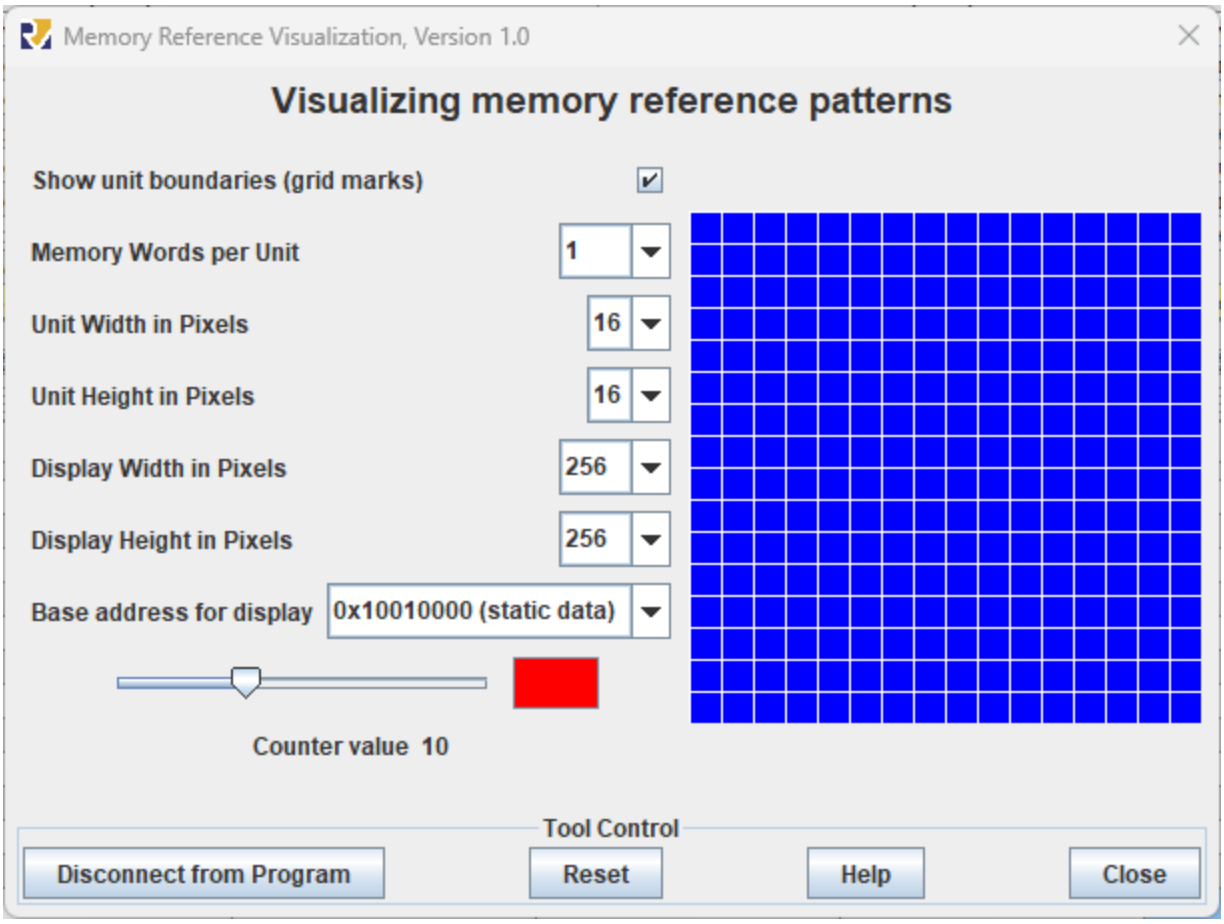
```
#############################################################################
####
#  Row-major order traversal of 16 x 16 array of words.
#  Pete Sanderson
#  31 March 2007
#
#  To easily observe the row-oriented order, run the Memory Reference
#  Visualization tool with its default settings over this program.
#  You may, at the same time or separately, run the Data Cache Simulator
#  over this program to observe caching performance.  Compare the results
#  with those of the column-major order traversal algorithm.
#
#  The C/C++/Java-like equivalent of this MIPS program is:
#     int size = 16;
#     int[size][size] data;
#     int value = 0;
#     for (int row = 0; row < size; row++) {
#        for (int col = 0; col < size; col++) }
#           data[row][col] = value;
#           value++;
#        }
#     }
#
#  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change
this,
```

```
#        three statements need to be changed.
#        1. The array storage size declaration at "data:" needs to be
changed from
#           256 (which is 16 * 16) to #columns * #rows.
#        2. The "li" to initialize $t0 needs to be changed to new #rows.
#        3. The "li" to initialize $t1 needs to be changed to new
#columns.
#
         .data
data:    .word     0 : 256       # storage for 16x16 matrix of words
         .text
         li       t0, 16        # $t0 = number of rows
         li       t1, 16        # $t1 = number of columns
         la              a0, data
         mv     s0, zero     # $s0 = row counter
         mv     s1, zero     # $s1 = column counter
         mv     t2, zero     # $t2 = the value to be stored
#  Each loop iteration will store incremented $t1 value into next element
of matrix.
#  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
#  Note: no attempt is made to optimize runtime performance!
loop:    mul      s2, s0, t1       # $s2 = row * #cols  (two-instruction
sequence)
         add      s2, s2, s1  # $s2 += column counter
         slli      s2, s2, 2    # $s2 *= 4 (shift left 2 bits) for byte
offset
         add  s2, a0, s2
         sw       t2, 0(s2) # store the value in matrix element
         addi     t2, t2, 1    # increment value to be stored
#  Loop control: If we increment past last column, reset column counter
and increment row counter
#                If we increment past last row, we're finished.
         addi    s1, s1, 1    # increment column counter
         bne      s1, t1, loop # not at end of row so loop back
         mv     s1, zero     # reset column counter
         addi     s0, s0, 1    # increment row counter
         bne      s0, t0, loop # not at end of matrix so loop back
#  We're finished traversing the matrix.
         li       a7, 10        # system service 10 is exit
         ecall                 # we are outta here.
```

Kết quả:


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close



1. Ảnh hưởng của Block Size (Kích thước khối):

- *Block size nhỏ:*
  - Tương tự như trường hợp duyệt theo cột, khi một phần tử được truy cập lần đầu tiên và không có trong cache (cache miss), một khối dữ liệu nhỏ xung quanh phần tử đó sẽ được nạp vào cache.
  - Tuy nhiên, điểm khác biệt quan trọng là trong duyệt theo hàng, các phần tử tiếp theo được truy cập (trong cùng một hàng) nằm liền kề trong bộ nhớ. Do đó, nếu block size đủ lớn để chứa nhiều phần tử liên tiếp của một hàng, các lần truy cập tiếp theo sẽ là cache hit.
  - Kết quả: Tỷ lệ cache hit sẽ tăng đáng kể so với duyệt theo cột khi block size đủ lớn để tận dụng tính cục bộ không gian theo hàng. Block size càng lớn (nhưng không vượt quá kích thước của một hàng hoặc các đặc tính cache khác), tỷ lệ cache hit càng cao trong mỗi hàng.
- *Block size lớn:*
  - Nếu block size lớn hơn kích thước của một vài phần tử liên tiếp trong một hàng, một lần nạp khối có thể mang vào cache nhiều dữ liệu sẽ được sử dụng ngay trong các lần truy cập tiếp theo của cùng hàng đó.
  - Ví dụ, nếu block size là 64 bytes, nó có thể chứa 16 words (64 bytes / 4 bytes/word), tức là toàn bộ một hàng của mảng. Trong trường hợp này, lần truy cập đầu tiên vào một hàng sẽ gây ra cache miss, nhưng 15 lần truy cập tiếp theo vào cùng hàng đó sẽ là cache hit (giả sử khối đó không bị loại bỏ trước khi được sử

dụng hết).
    - Kết quả: Tỷ lệ cache hit rất cao khi block size phù hợp với kích thước của dữ liệu được truy cập tuần tự.

2. Ảnh hưởng của Number of Blocks (Số lượng khối):

- *Số lượng khối nhỏ:*
    - Khi số lượng khối nhỏ, cache có ít chỗ để lưu trữ các khối dữ liệu khác nhau. Khi chương trình chuyển sang hàng tiếp theo, các khối dữ liệu của hàng trước đó có thể đã bị loại bỏ nếu không được sử dụng lại gần đây.
    - Tuy nhiên, do tính cục bộ thời gian (các phần tử trong cùng một hàng được truy cập liên tiếp), một số khối của hàng hiện tại sẽ được sử dụng nhiều lần trước khi chuyển sang hàng khác.
    - Kết quả: Tỷ lệ cache hit có thể bị giảm nếu số lượng khối quá nhỏ, dẫn đến việc các khối cần thiết bị loại bỏ quá sớm.
- *Số lượng khối lớn:*
    - Với số lượng khối lớn hơn, cache có thể lưu trữ nhiều hàng đã được truy cập gần đây. Khi chương trình chuyển sang hàng tiếp theo, nếu các khối của hàng đó vẫn còn trong cache (do không bị ghi đè bởi các truy cập khác), tỷ lệ cache hit sẽ cao hơn.
    - Đặc biệt, nếu kích thước cache đủ lớn để chứa toàn bộ hoặc một phần đáng kể của mảng, việc tăng số lượng khối sẽ giúp giảm thiểu các cache miss do xung đột dung lượng khi truy cập các hàng khác nhau.
    - Kết quả: Tỷ lệ cache hit cao hơn so với số lượng khối nhỏ, đặc biệt khi kích thước cache đủ lớn để tận dụng tính cục bộ thời gian giữa các hàng.

Tóm lại:

Đối với chương trình duyệt mảng theo thứ tự hàng này:

- Block size: Block size đóng vai trò then chốt trong việc tận dụng tính cục bộ không gian theo hàng. Một block size đủ lớn (ví dụ, chứa vài phần tử liên tiếp hoặc thậm chí toàn bộ một hàng) sẽ dẫn đến tỷ lệ cache hit rất cao trong quá trình duyệt mỗi hàng.
- Number of Blocks: Số lượng khối đủ lớn giúp duy trì các khối dữ liệu của các hàng đã được truy cập trong cache, giảm thiểu cache miss khi chuyển sang các hàng tiếp theo và cải thiện hiệu suất tổng thể.

So sánh với duyệt theo cột (Column-major order):

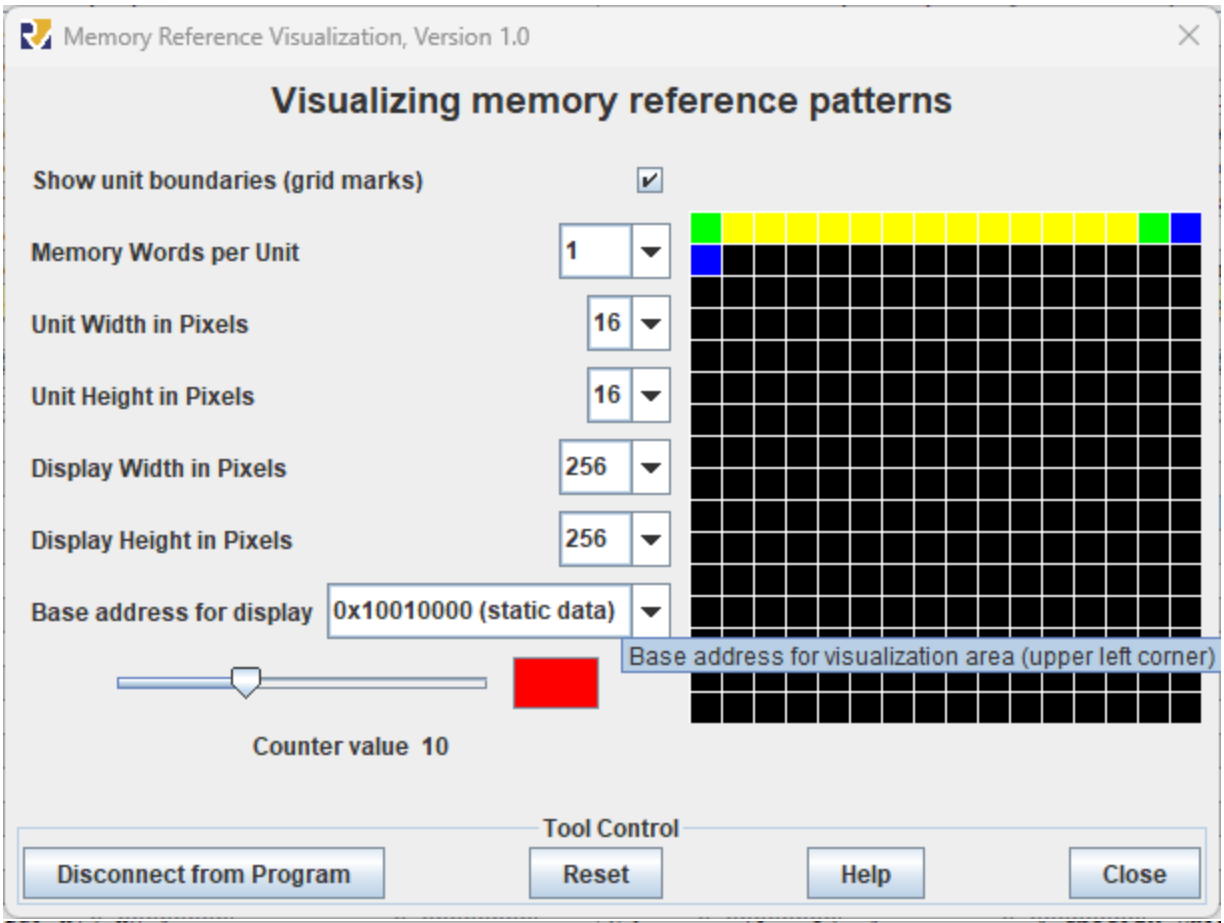
Như đã phân tích ở lần trước, duyệt theo cột có tỷ lệ cache hit thấp hơn đáng kể. Lý do là vì các phần tử được truy cập liên tiếp cách xa nhau trong bộ nhớ, làm giảm hiệu quả của việc nạp một khối vào cache. Duyệt theo hàng tận dụng tốt hơn tính cục bộ không gian của dữ liệu mảng được lưu trữ tuần tự trong bộ nhớ, dẫn đến hiệu suất cache cao hơn nhiều.

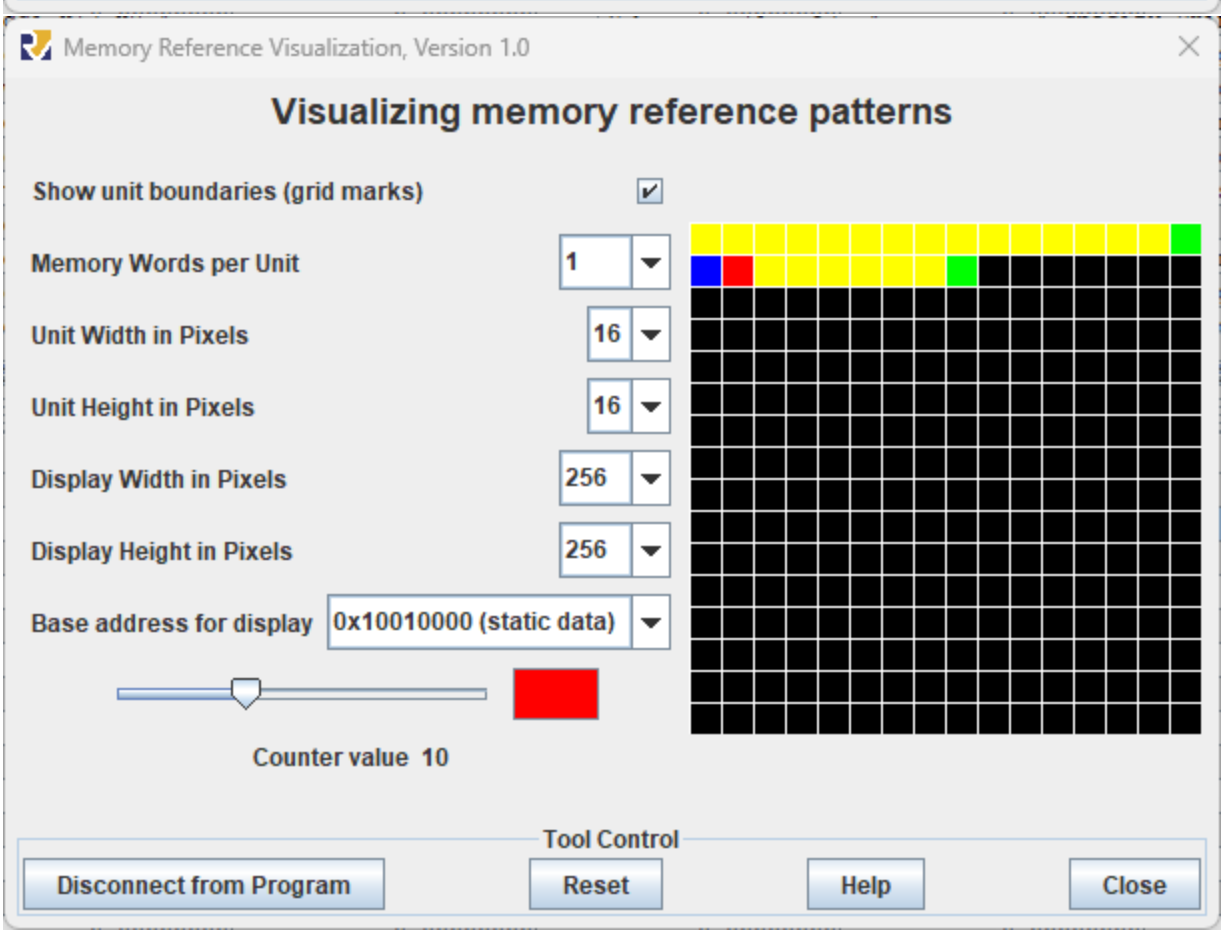
Đoạn mã:

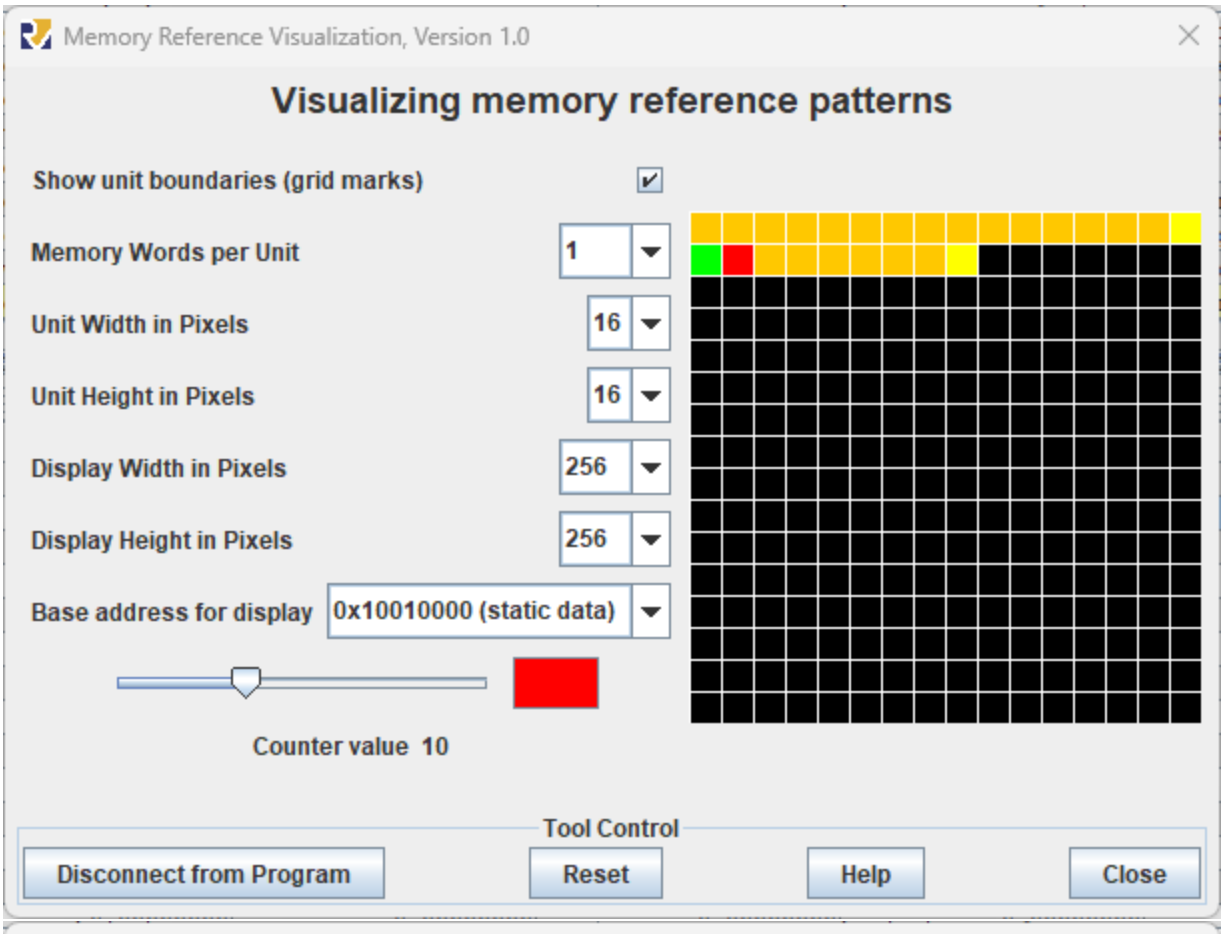
```
# Compute first twelve Fibonacci numbers and put in array, then print
      .data
fibs: .word   0 : 16        # "array" of 12 words to contain fib values
size: .word  16             # size of "array"
      .text
      la   t0, fibs        # load address of array
      la   t5, size        # load address of size variable
      lw   t5, 0(t5)      # load array size
      li   t2, 1           # 1 is first and second Fib. number
      #add.d $f0, $f2, $f4
```

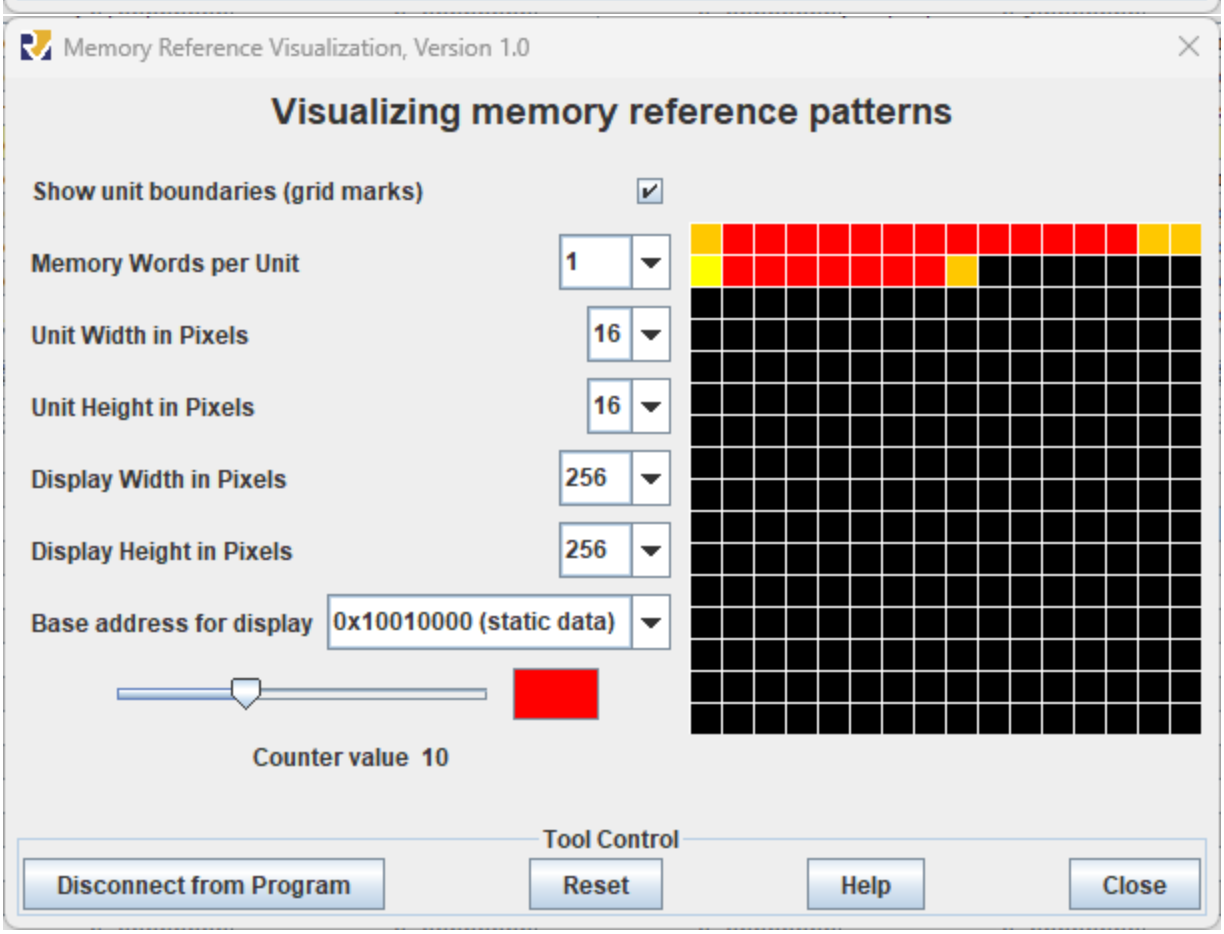
```
      sw   t2, 0(t0)      # F[0] = 1
      sw   t2, 4(t0)      # F[1] = F[0] = 1
      addi t1, t5, -2     # Counter for loop, will execute (size-2) times
loop: lw   t3, 0(t0)      # Get value from array F[n]
      lw   t4, 4(t0)      # Get value from array F[n+1]
      add  t2, t3, t4    # $t2 = F[n] + F[n+1]
      sw   t2, 8(t0)      # Store F[n+2] = F[n] + F[n+1] in array
      addi t0, t0, 4      # increment address of Fib. number source
      addi t1, t1, -1     # decrement loop counter
      bgtz t1, loop        # repeat if not finished yet.
      la   a0, fibs        # first argument for print (array)
      add  a1, zero, t5  # second argument for print (size)
      jal  print            # call print routine.
      li   a7, 10          # system call for exit
      ecall               # we are out of here.

#########  routine to print the numbers on one line.

      .data
space:.asciz  " "          # space to insert between numbers
head: .asciz  "The Fibonacci numbers are:\n"
      .text
print:add  t0, zero, a0  # starting address of array
      add  t1, zero, a1  # initialize loop counter to array size
      la   a0, head        # load address of print heading
      li   a7, 4           # specify Print String service
      ecall               # print heading
out:  lw   a0, 0(t0)      # load fibonacci number for syscall
      li   a7, 1           # specify Print Integer service
      ecall               # print fibonacci number
      la   a0, space       # load address of spacer for syscall
      li   a7, 4           # specify Print String service
      ecall               # output string
      addi t0, t0, 4      # increment address
      addi t1, t1, -1     # decrement loop counter
      bgtz t1, out         # repeat if not finished
      jr   ra              # return
```

Kết quả:


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Base address for visualization area (upper left corner)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close


Memory Reference Visualization, Version 1.0
Visualizing memory reference patterns
Show unit boundaries (grid marks)
Memory Words per Unit
1
Unit Width in Pixels
16
Unit Height in Pixels
16
Display Width in Pixels
256
Display Height in Pixels
256
Base address for display
0x10010000 (static data)
Counter value 10
Tool Control
Disconnect from Program
Reset
Help
Close

Chương trình này thực hiện hai giai đoạn chính liên quan đến bộ nhớ:

1. *Ghi các số Fibonacci vào mảng fibs: Chương trình tính toán 10 số Fibonacci tiếp theo (sau hai số khởi tạo là 1 và 1) và lưu trữ chúng tuần tự vào mảng fibs.*
2. *Đọc và in các số Fibonacci từ mảng fibs: Chương trình sau đó đọc các số đã lưu trữ trong mảng fibs và in chúng ra màn hình.*

Ta sẽ phân tích ảnh hưởng của cấu trúc cache lên từng giai đoạn này. Mảng fibs chứa 12 words, tương đương với 48 bytes (12 * 4 bytes/word).

Giai đoạn 1: Ghi các số Fibonacci vào mảng fibs

Trong vòng lặp chính (loop), chương trình thực hiện các bước sau:

1. Đọc hai số Fibonacci liền kề trước đó từ mảng (lw t3, 0(t0) và lw t4, 4(t0)).
2. Tính số Fibonacci tiếp theo (add t2, t3, t4).
3. Ghi số Fibonacci vừa tính vào vị trí kế tiếp trong mảng (sw t2, 8(t0)).
4. Tăng con trỏ t0 lên 4 bytes để trỏ đến vị trí của hai số Fibonacci tiếp theo sẽ được đọc.

- **Ảnh hưởng của Block Size:**
  - Block size nhỏ: Khi chương trình ghi F[0] và F[1], các khối chứa các vị trí này sẽ được đưa vào cache. Khi tính toán và ghi các số Fibonacci tiếp theo, nếu block size nhỏ, mỗi lần ghi (sw) có thể gây ra một cache miss nếu khối chứa vị trí ghi chưa được nạp vào cache. Tuy nhiên, do các lần ghi diễn ra tuần tự và gần nhau trong bộ nhớ, sau lần miss đầu tiên, các lần ghi tiếp theo có thể là hit nếu các vị trí ghi nằm trong cùng một khối đã được nạp.
  - Block size lớn: Nếu block size đủ lớn để chứa nhiều phần tử liên tiếp của mảng fibs, khi khối chứa F[0] và F[1] được nạp, các vị trí bộ nhớ cho các số Fibonacci tiếp theo cũng có thể được đưa vào cache. Điều này sẽ làm giảm số lượng cache miss khi ghi các số tiếp theo. Ví dụ, nếu block size là 16 bytes, nó có thể chứa 4 words. Sau khi F[0] và F[1] được ghi, khối này có thể chứa sẵn vị trí cho F[2] và F[3].
- **Ảnh hưởng của Number of Blocks:**
  - Vì giai đoạn ghi diễn ra tuần tự trên một vùng nhớ nhỏ (48 bytes), số lượng khối thường không có ảnh hưởng lớn đến tỷ lệ cache hit trong giai đoạn này, miễn là cache có đủ chỗ để chứa các khối đang được truy cập. Nếu số lượng khối quá nhỏ, có thể xảy ra xung đột nếu các vùng nhớ khác được truy cập đồng thời, nhưng trong đoạn mã này, các truy cập chủ yếu tập trung vào mảng fibs.

Giai đoạn 2: Đọc và in các số Fibonacci từ mảng fibs (trong hàm print)

Trong hàm print, chương trình đọc tuần tự từng số Fibonacci từ mảng fibs và in ra.

- **Ảnh hưởng của Block Size:**
  - Block size nhỏ: Khi chương trình đọc F[0] lần đầu tiên (lw a0, 0(t0)), khối chứa nó sẽ được nạp vào cache. Nếu block size nhỏ, chỉ một vài số Fibonacci đầu tiên có thể nằm trong khối này. Các lần đọc tiếp theo sẽ tiếp tục nạp các khối chứa các số Fibonacci kế tiếp. Tuy nhiên, do tính tuần tự trong việc đọc, sau lần miss đầu tiên cho mỗi khối, các lần đọc các phần tử còn lại trong khối đó sẽ là hit.
  - Block size lớn: Nếu block size đủ lớn để chứa nhiều số Fibonacci liên tiếp (ví dụ, 16 bytes chứa 4 số), một lần nạp khối có thể phục vụ cho nhiều lần đọc tiếp theo trong cùng một khối. Điều này sẽ làm tăng đáng kể tỷ lệ cache hit trong quá trình in. Ví dụ, nếu block size là 16 bytes, lần đọc F[0] sẽ nạp khối chứa F[0],

F[1], F[2], F[3]. Ba lần đọc tiếp theo sẽ là cache hit.

- Ảnh hưởng của Number of Blocks:
    - Vì quá trình in đọc toàn bộ mảng fibs một lần duy nhất theo thứ tự tuần tự, số lượng khối có vai trò quan trọng trong việc giữ các khối đã được đọc trong cache nếu có các hoạt động truy cập bộ nhớ khác xen kẽ (điều này không xảy ra rõ ràng trong đoạn mã này, nhưng có thể xảy ra trong một chương trình lớn hơn). Nếu số lượng khối đủ lớn để chứa toàn bộ mảng fibs (hoặc phần lớn của nó), sau lần truy cập đầu tiên vào mỗi khối, các lần truy cập tiếp theo vào các số Fibonacci khác sẽ có khả năng cao là hit nếu các khối này vẫn còn trong cache.

Tổng kết:

Đối với chương trình tính và in số Fibonacci này:

- Block size: Block size có ảnh hưởng đáng kể đến tỷ lệ cache hit, đặc biệt trong giai đoạn đọc và in. Một block size phù hợp (không quá nhỏ và không quá lớn so với kích thước của dữ liệu được truy cập tuần tự) sẽ giúp tận dụng tính cục bộ không gian, làm tăng số lượng cache hit khi truy cập các phần tử lân cận trong mảng.
- Number of Blocks: Số lượng khối ảnh hưởng đến khả năng giữ lại các khối dữ liệu đã được truy cập trong cache. Với một chương trình nhỏ và truy cập tuần tự như thế này, miễn là số lượng khối đủ để chứa các khối đang được truy cập, ảnh hưởng của nó có thể không quá lớn. Tuy nhiên, trong các chương trình phức tạp hơn với nhiều vùng nhớ được truy cập, số lượng khối lớn hơn sẽ giúp giảm cache miss do xung đột.

Trong trường hợp này, vì cả giai đoạn ghi và đọc mảng fibs đều diễn ra tuần tự trên một vùng nhớ nhỏ, một block size vừa phải (ví dụ, chứa 2-4 words) sẽ mang lại lợi ích đáng kể về tỷ lệ cache hit. Số lượng khối cần đủ để chứa các khối này trong quá trình truy cập.