ABSTRACT

1) CUDA: Compute Unified Device Architecture

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs (Graphics Processing Units). It allows developers to utilize the massive parallel processing power of GPUs to accelerate computationally intensive applications.

We have used google collab for simulation of matrix multiplication CUDA programming model based on the C programming language,

2) Configuring the data cache in the given specification on RIPES.

RIPES (RISC-V Processor Ecosystem) is a simulator that allows us to design and test RISC-V processor architectures, including cache configurations. It provides a convenient platform to experiment with different cache configurations and evaluate their performance without the need for actual hardware. This allows for quick iteration and optimization of the cache design before implementing it in actual hardware.

PROBLEM STATEMENTS

CUDA:

You are given two large matrices A and B, represented as two-dimensional arrays of size N x N, where N is a positive integer. You need to implement a CUDA program to perform matrix multiplication of A and B, and store the result in a new matrix C, also of size N x N.

RIPES:

For the assignment, you are expected to configure the data cache in the given specification on RIPES and attach screenshots of the cache simulated. Run the matrix multiplier code to simulate each data cache configuration and report cache size, hit rate and writebacks.

1. Configure a 64-entry 8-word direct mapped cache.
2. Configure a 16-entry 4-word 4-way set-associative cache.
3. Configure a 16-entry 2-word 4-way set-associative cache with write-through.
4. Configure a 64-entry 8-word fully associative cache with Least Recently Used replacement policy and report the numbers. Change the replacement policy to Random and report the numbers for the same cache.
5. Configure a 32-entry 2-word direct-mapped cache and plot a graph using the plot configuration with numerator as Hits and denominator as Access count. Explain why the number of hits increases and decreases back down before increasing again.
6. Configure a 16-entry 4-word 2-way set-associative cache with write-back and write allocate and report the numbers. For the same configuration of cache, use writethrough and no write allocate and report the numbers. Explain the differences between the two cache configurations

OBJECTIVE

1) CUDA:
   The objective is to demonstrate how to use CUDA to perform matrix multiplication using GPU parallelism. The code uses the GPU to perform matrix multiplication on two input matrices A and B of size N x N and stores the result in an output matrix C of size N x N.

   The code demonstrates how to use CUDA's memory allocation functions to allocate memory on the GPU, copy data between CPU and GPU memory using cudaMemcpy function, and organize computation using two-dimensional thread blocks and grids to achieve better memory access patterns. The program also generates two input matrices with random values and prints them on the console.

2) RIPES:
   The objective is to understand the impact of different cache configurations on the performance of a matrix multiplication code. We have configured different cache sizes and replacement policies, and measure the hit rate, write-backs, and access counts for each configuration. Looking at the graph of hit rate vs access count for a particular cache configuration and note the observation. Finally compared the performance of two cache configurations with different write policies and write allocation strategies. By performing these tasks, the objective is to gain insights into how cache configurations affect the performance of memory-intensive applications such as matrix multiplication.

METHODOLOGY

CUDA:

The code starts by allocating memory for the input matrices (h_A and h_B) and the output matrix (h_C) on the CPU (Central Processing Unit) using malloc function. The matrices are then generated with random values using the rand() function. The matrix multiplication operation is then performed on the GPU using the matrixMul() function. This function first allocates memory on the GPU using cudaMalloc() function, and then copies the input matrices from CPU memory to GPU memory using cudaMemcpy() function. The matrix multiplication is performed by launching the matrixMulKernel() CUDA kernel, which computes the product of two matrices and stores the result in the output matrix on the GPU. The kernel uses thread blocks and grids to parallelize the computation. The computed result matrix is then copied back from the GPU memory to the CPU memory using cudaMemcpy() function.

Finally, the result matrix is printed on the console, and the memory allocated on the CPU and GPU is freed using the free() and cudaFree() functions, respectively.

RIPES

We have used same matrix multiplication code in C language. Input the code to in the editor window of ripes software compile and run. Go to Cache window and set presets according to problem statement and configure to cache memory of different specifications and approaches such as writeback or writethrough, going through the plot of the configuration to observe the hits and misses.

IMPLEMENTATION

code:

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void matrixMulKernel(int* A, int* B, int* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        int sum = 0;
        for (int i = 0; i < N; i++) {
            sum += A[row * N + i] * B[i * N + col];
        }
        C[row * N + col] = sum;
    }
}

void matrixMul(int* h_A, int* h_B, int* h_C, int N) {
    int *d_A, *d_B, *d_C;
    size_t size = N * N * sizeof(int);

    // Allocate memory on the GPU
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy input matrices to the GPU memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Set the kernel dimensions and launch the kernel
    dim3 dimBlock(32, 32);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) / dimB
lock.y);
    matrixMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

    // Copy the result matrix from GPU memory to CPU memory
```

```
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free the GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

int main() {
    int N = 5;  // reduced matrix size

    // Allocate memory for the input and output matrices on the CPU
    int *h_A, *h_B, *h_C;
    size_t size = N * N * sizeof(int);
    h_A = (int*)malloc(size);
    h_B = (int*)malloc(size);
    h_C = (int*)malloc(size);

    // Populate the input matrices with random values
    printf("Matrix A:\n");
    for (int i = 0; i < N * N; i++) {
        h_A[i] = rand() % 10;
        printf("%d ", h_A[i]);
        if ((i+1) % N == 0) printf("\n");
    }
    printf("\n");

    printf("Matrix B:\n");
    for (int i = 0; i < N * N; i++) {
        h_B[i] = rand() % 10;
        printf("%d ", h_B[i]);
        if ((i+1) % N == 0) printf("\n");
    }
    printf("\n");

    // Perform matrix multiplication using CUDA
    matrixMul(h_A, h_B, h_C, N);

    // Print the result matrix
    printf("Result Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", h_C[i * N + j]);
        }
```

```
            printf("\n");
        }

         // Free the CPU memory
        free(h_A);
        free(h_B);
        free(h_C);

        return 0;
}
```

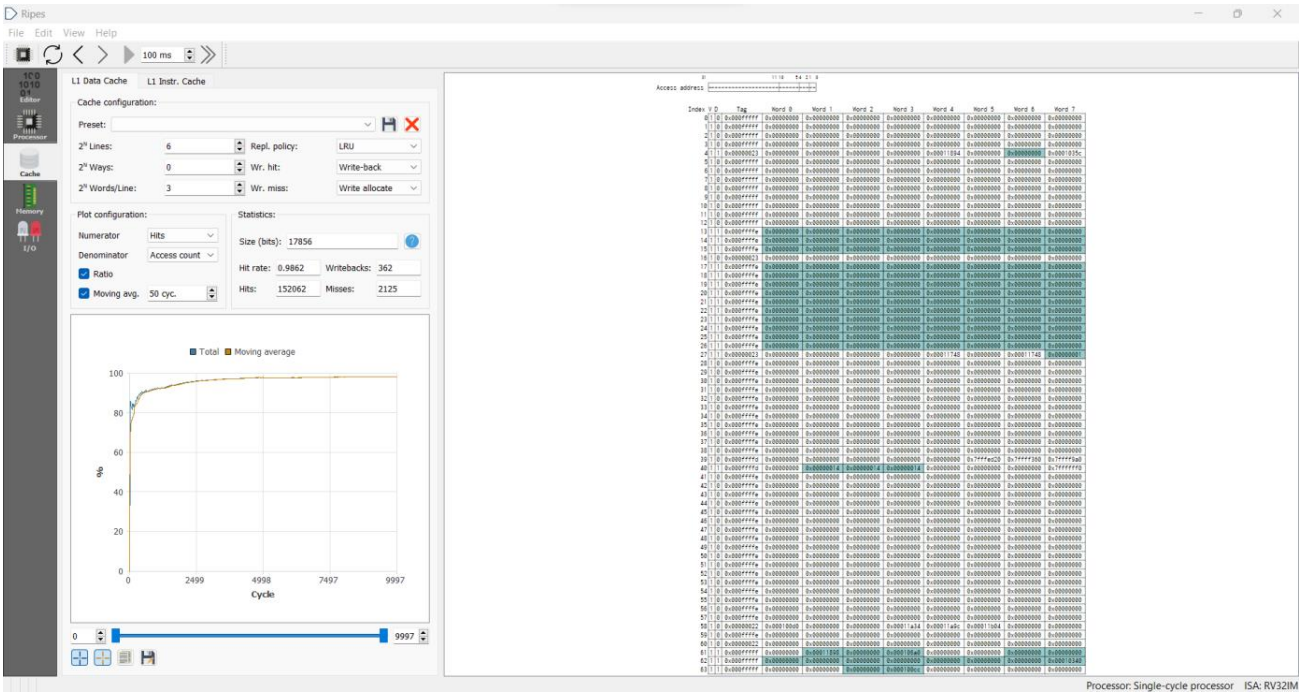
RESULTS

CUDA (OUTPUT) :

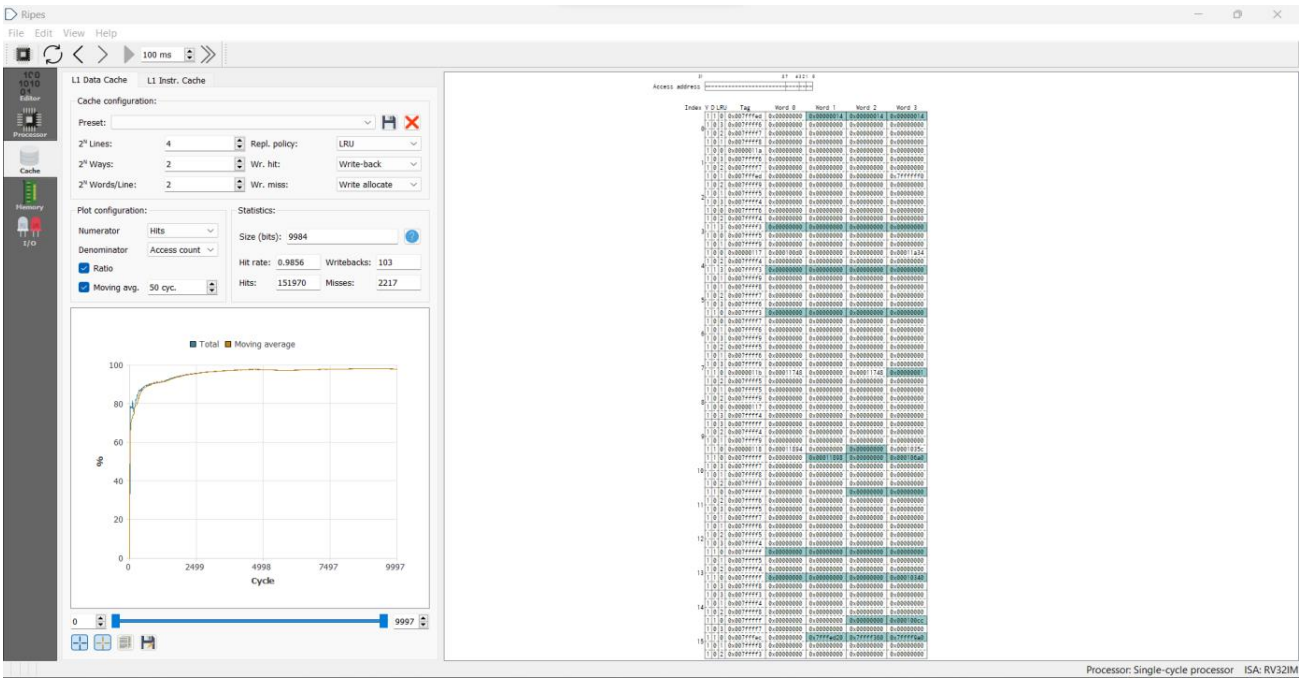
```
Matrix A:
3 6 7 5 3
5 6 2 9 1
2 7 0 9 3
6 0 6 2 6
1 8 7 9 2

Matrix B:
0 2 3 7 5
9 2 2 8 9
7 3 6 1 2
9 3 1 9 4
7 8 4 5 0

Result Matrix C:
169 78 80 136 103
156 63 52 171 119
165 69 41 166 109
102 84 80 96 50
216 82 78 169 127
```

RIPES:

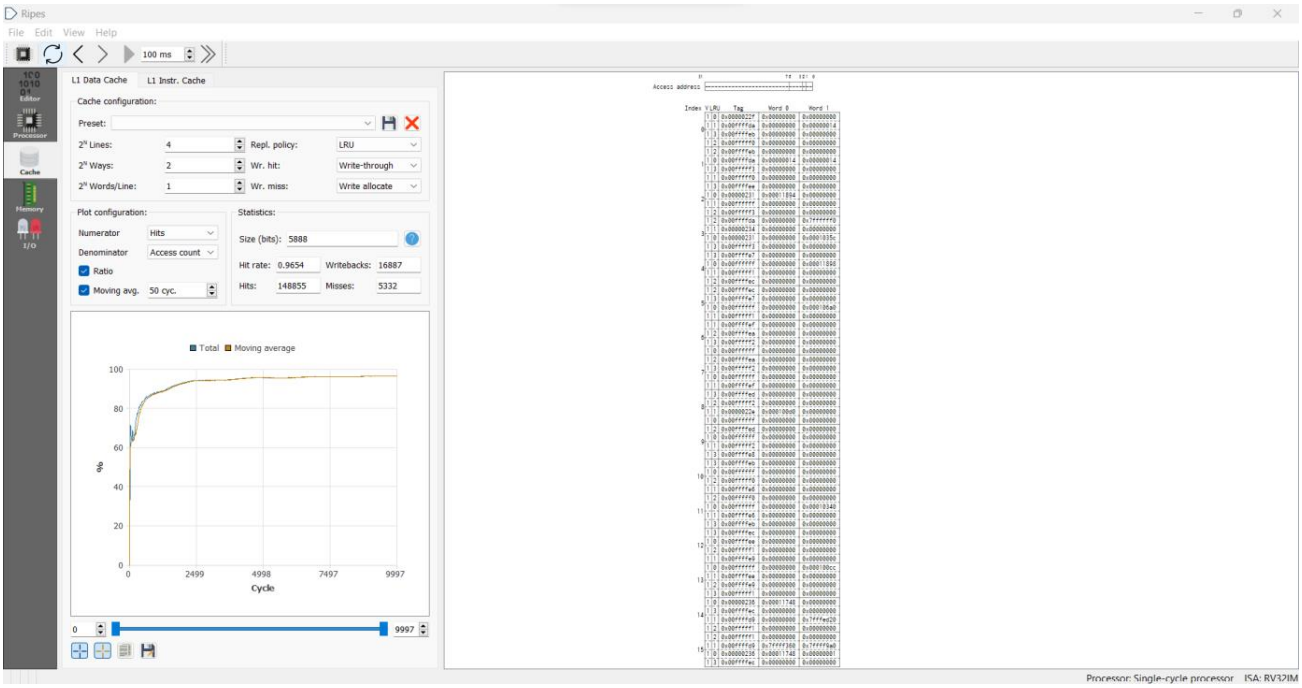
Q1)


Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 6
Repl. policy: LRU
2^n Ways: 0
Wr. hit: Write-back
2^n Words/Line: 3
Wr. miss: Write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 17856
Hit rate: 0.9862
Writebacks: 362
Hits: 152062
Misses: 2125
Total
Moving average
%
Cycle
0
2499
4998
7497
9997
Processor: Single-cycle processor ISA: RV32IM

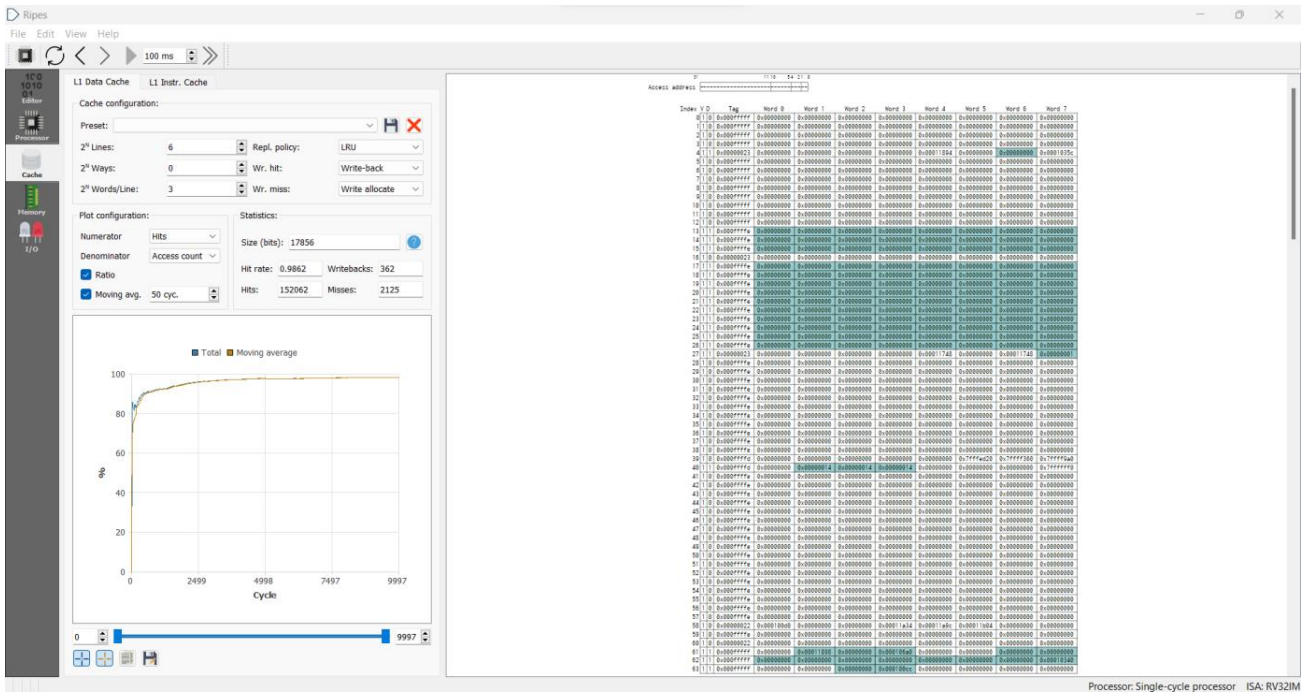
Q2)


Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 4
2^n Ways: 2
2^n Words/Line: 2
Repl. policy: LRU
Wr. hit: Write-back
Wr. miss: Write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 9984
Hit rate: 0.9856
Writebacks: 103
Hits: 151970
Misses: 2217
Total
Moving average
%
Cycle
Processor: Single-cycle processor ISA: RV32IM

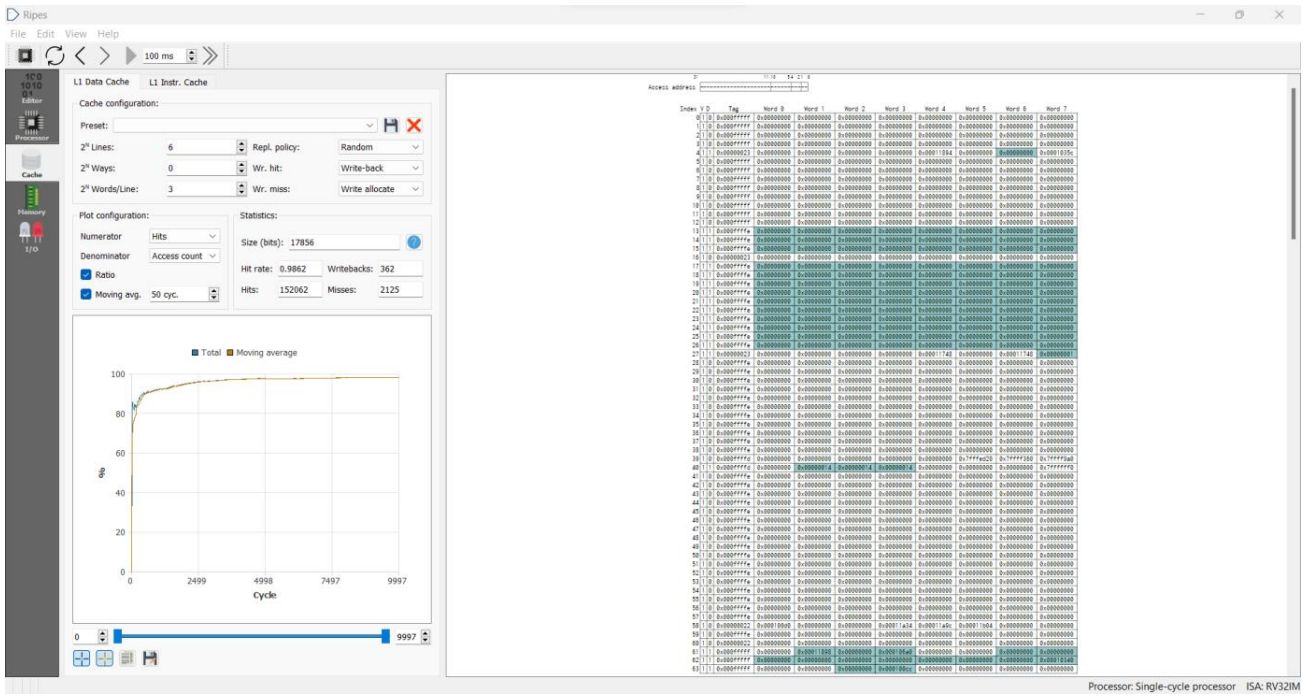
Q3)


Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 4
2^n Ways: 2
2^n Words/Line: 1
Repl. policy: LRU
Wr. hit: Write-through
Wr. miss: Write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 5888
Hit rate: 0.9654
Writebacks: 16887
Hits: 148855
Misses: 5332
Total
Moving average
%
Cycle
Processor: Single-cycle processor ISA: RV32IM

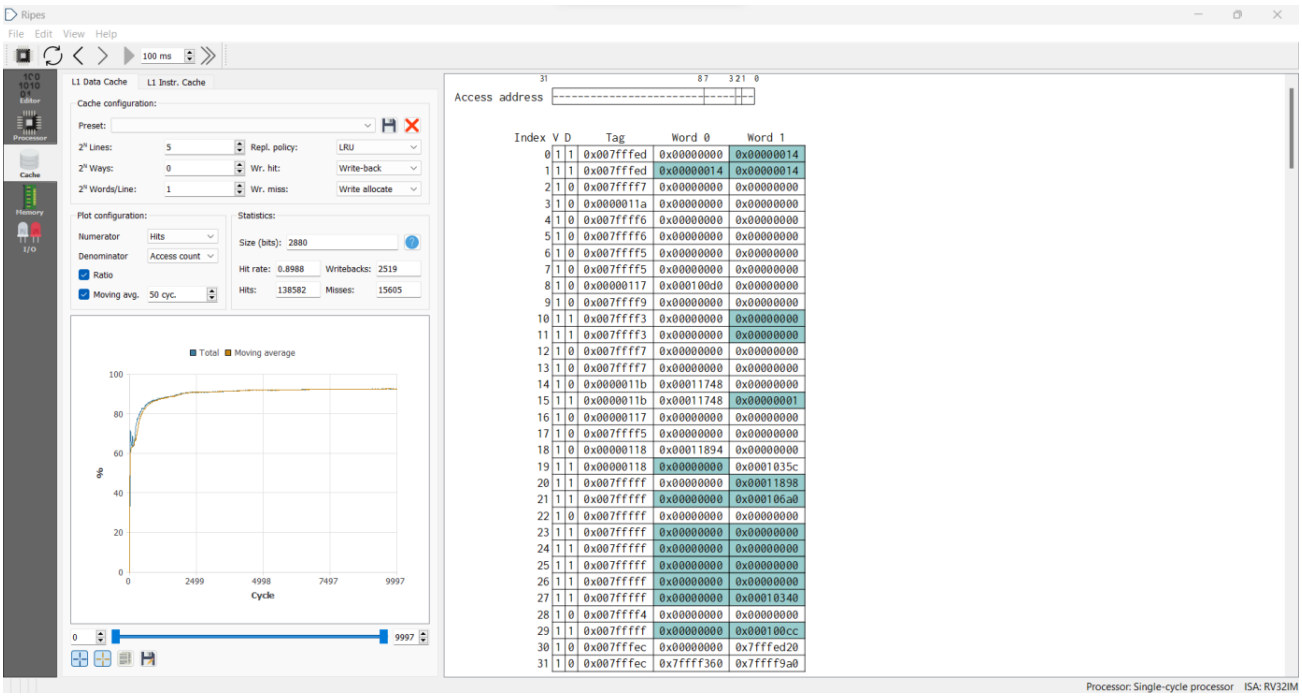
4a)


Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 6
2^n Ways: 0
2^n Words/Line: 3
Repl. policy: LRU
Wr. hit: Write-back
Wr. miss: Write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 17856
Hit rate: 0.9862
Writebacks: 362
Hits: 152062
Misses: 2125
Total
Moving average
%
Cycle
0
2499
4998
7497
9997
Processor: Single-cycle processor ISA: RV32IM

4b)

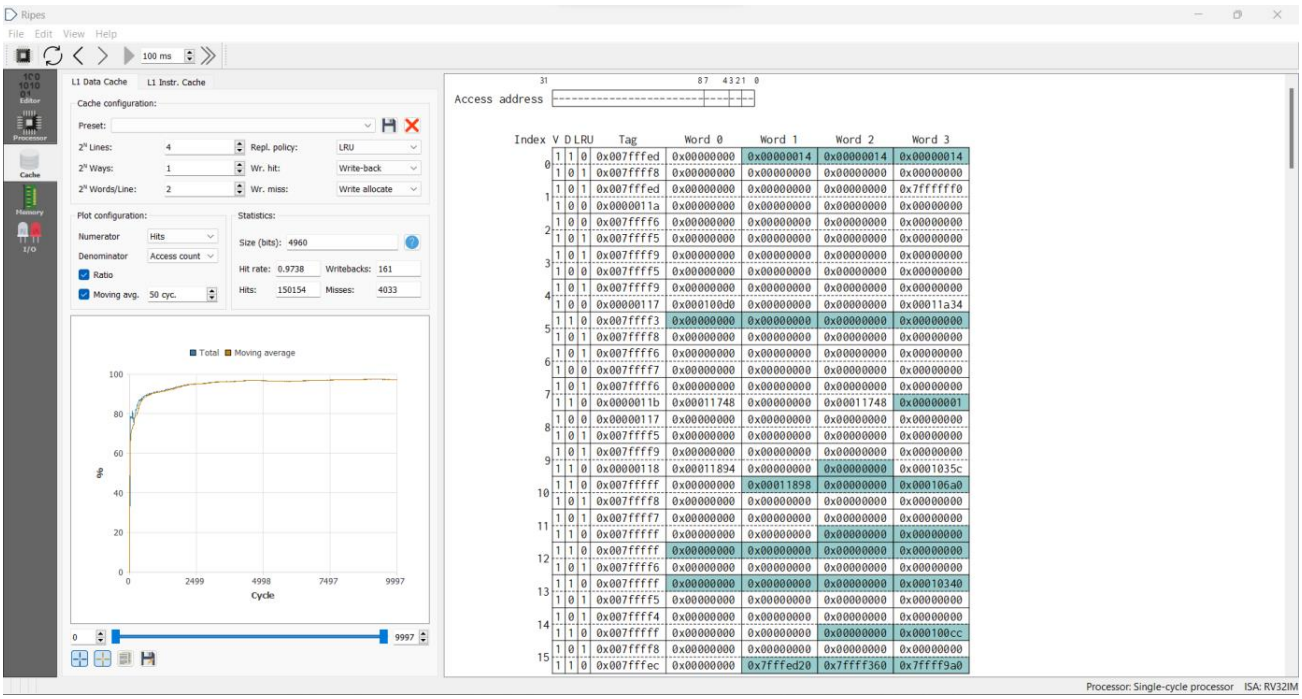

Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 6
2^n Ways: 0
2^n Words/Line: 3
Repl. policy: Random
Wr. hit: Write-back
Wr. miss: Write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 17856
Hit rate: 0.9862
Writebacks: 362
Hits: 152062
Misses: 2125
Total
Moving average
%
Cycle
0
2499
4998
7497
9997
Processor: Single-cycle processor ISA: RV32IM

5)

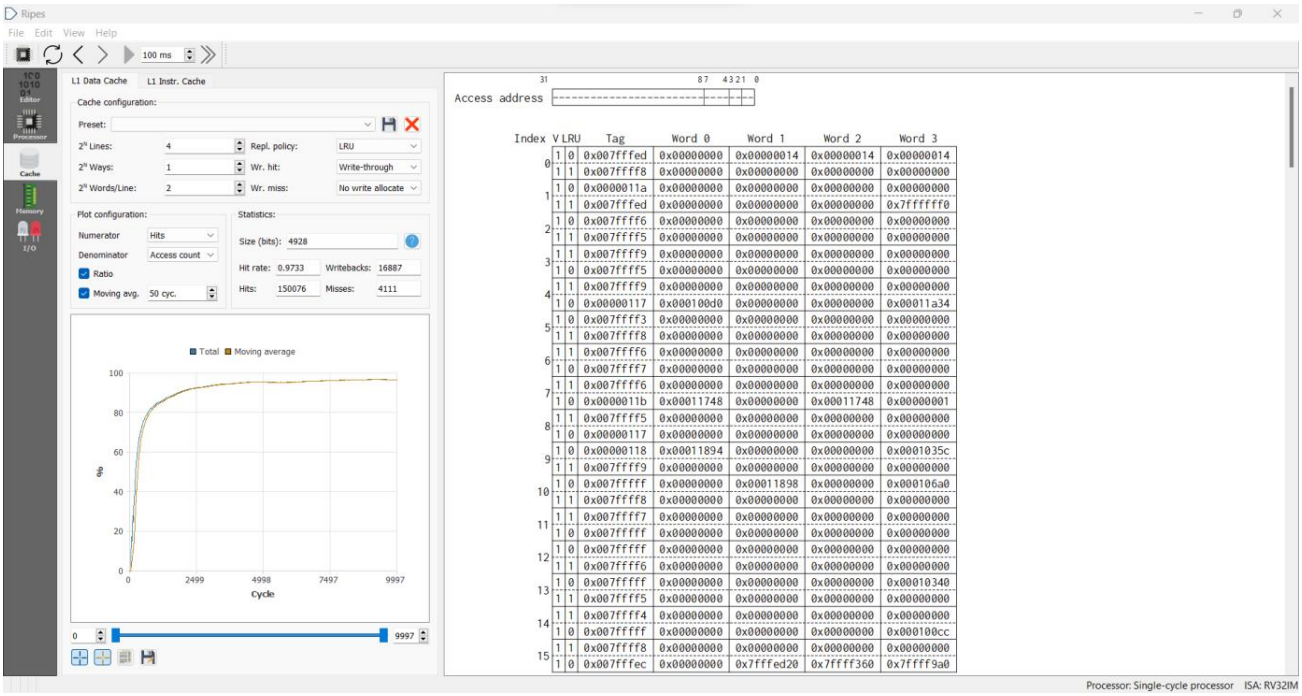


In a 32-entry 2-word direct-mapped cache, the number of hits increases and decreases back down before increasing again because of cache conflicts. When a program accesses memory blocks that map to the same cache line, the cache can only hold one of the blocks at a time, so it will evict one and fetch the other, resulting in a cache miss. As the program continues to access these memory blocks, the cache will alternate between holding one block and the other, resulting in alternating hits and misses on the cache line. Therefore, the number of hits may increase up to a maximum of 16 per cache line (since there are only 16 cache lines available for each set of two memory blocks that map to the same cache line), but will eventually decrease due to evictions and cache misses, before increasing again when the program accesses the memory blocks that were evicted earlier.

6a)


Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 4
Repl. policy: LRU
2^n Ways: 1
Wr. hit: Write-back
2^n Words/Line: 2
Wr. miss: Write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 4960
Hit rate: 0.9738
Writebacks: 161
Hits: 150154
Misses: 4033
Total
Moving average
Cycle
Access address
Index V D LRU Tag Word 0 Word 1 Word 2 Word 3
Processor: Single-cycle processor ISA: RV32IM

6b)


Ripes
File Edit View Help
100 ms
L1 Data Cache
L1 Instr. Cache
Cache configuration:
Preset:
2^n Lines: 4
Repl. policy: LRU
2^n Ways: 1
Wr. hit: Write-through
2^n Words/Line: 2
Wr. miss: No write allocate
Plot configuration:
Numerator Hits
Denominator Access count
Ratio
Moving avg. 50 cyc.
Statistics:
Size (bits): 4928
Hit rate: 0.9733
Writebacks: 16887
Hits: 150076
Misses: 4111
Total
Moving average
Cycle
Access address
Index V LRU Tag Word 0 Word 1 Word 2 Word 3
Processor: Single-cycle processor ISA: RV32IM

Difference between two cache configurations:

In a Cache with write-through and no-write-allocate policies, write operations are immediately written both to the cache and main memory. No-write-allocate means that when a write operation misses in the cache, the write is directly performed in main memory without bringing the block into the cache.

CONCLUSION

We notice that the CUDA kernel can be implemented to perform various operations on the GPU making use of memory management, taking advantage of the parallel processing capabilities of the GPU. Minimizing the data transfers between CPU and GPU, and effectively utilizing the CUDA threads and blocks to achieve maximum throughput and performance.

Different data caches can be configured using RIPES, of different sizes for the cache and different approaches such as direct mapping, set associative and fully associative. Difference in the hits, hit rate, misses, miss rates and other parameters as well is studied and compared.