CSE530 Assignment 0001 tutorial

> In this notebook exercise we are trying to get you familiar with simulator infrastructure and get you started with cache performance analysis using matrix multiplication kernel as example.

Step1: Download pintool package from the intel software center

```
!wget https://software.intel.com/sites/landingpage/pintool/downloads/pin-3.18-98332-g
```

Step 2: Untar the gz file

```
%cd /content/
!tar -zxvf /tmp/pin-3.18-98332-gaebd7b1e6-gcc-linux.tar.gz
```

Step 3: clone the simulator from Abhishek's github

```
%cd /content
!git clone https://github.com/abhishekk06/CachePerformanceOnMatMul.git
```

Step 4: Update the pinatrace.cpp file so that intel pin tool will start/stop recording memory access when start_roi()/stop_roi() is called

```
!cp CachePerformanceOnMatMul/pintool_script/pinatrace.cpp pin-3.18-98332-gaebd7b1e6-g
```

Step 5: Build the tool. Use lscpu to ensure the target architecture, if incase it is 32 bit, append the make command with TARGET=ia32

```
%cd /content/pin-3.18-98332-gaebd7b1e6-gcc-linux/source/tools/
!make all
```

Step 6: Setting up environment variable for the pintool.

```
%env PIN_ROOT=/content/pin-3.18-98332-gaebd7b1e6-gcc-linux/
!echo $PIN_ROOT
```

NOTE: Ensure your repo is clean by removing the bin and traces directories

```
%cd /content/CachePerformanceOnMatMul/
!rm -rf bin/
!rm -rf traces/
```

```
/content/CachePerformanceOnMatMul
```

Step 7: Compile your kernel by using the following g++ command.

NOTE: We recommend everyone to walk through the code in order to build the insight and for the better understanding.

```
!cd /content/CachePerformanceOnMatMul/
!mkdir -p bin/
!mkdir -p traces/
```

```
!echo Compiling Matmul_ijkalgo
!g++ -Wall src/matmul_ijkalgo.cpp -o bin/matmul_ijkalgo.o
```

Step 8(Optional): Our kernel demands matrices as input. In order to generate random 2D matrix of different size and sparsity, simulator provides a utility script. "random_matrix_generator" generates the files in normal representation and as well as the CSR representation. Ignore the CSR representation for now.

The python script requires size, output_file_name and sparsity as its input.

For our current example, we generated matrix of size 10 with sparsity 50% and named the output file as input_matrix.in

---

```
!python utils/random_matrix_generator.py --n 10 --dump input_matrix.in --sparsity 50
```

Step 9: Generate the traces using pintool on matrix multiplication kernel

```
!$PIN_ROOT/pin -t $PIN_ROOT/source/tools/ManualExamples/obj-intel64/pinatrace.so -- b
!mv pinatrace.out traces/ijk_traces.out
```

Step 10: Run Cache_simulator.py using the following script.

The default cache parameters are mentioned in Simulator/config/config_simple_multilevel

```
!source run_simulator.sh /content/CachePerformanceOnMatMul/traces
```

Step 11: Performing similar experiment with ikj matrix multiplication kernel.

```
%cd /content/CachePerformanceOnMatMul/
!rm -rf bin/
!rm -rf traces/

!mkdir -p bin/
!mkdir -p traces/
```

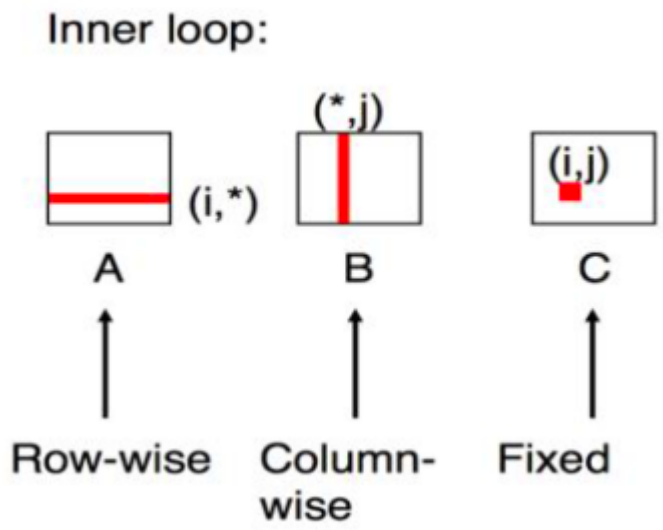
```
    /content/CachePerformanceOnMatMul
```

```
!echo Compiling Matmul_ikjalgo
!g++ -Wall src/matmul_ikjalgo.cpp -o bin/matmul_ikjalgo.o

!$PIN_ROOT/pin -t $PIN_ROOT/source/tools/ManualExamples/obj-intel64/pinatrace.so -- b
!mv pinatrace.out traces/ikj_traces.out

!source run_simulator.sh /content/CachePerformanceOnMatMul/traces
```

Cache Study - Deep Dive

By comparing the results in both benchmark, we can conclude that ikj kernel is performing matrix multplications in less number of cylces. The theory behind ikj being faster than ijk kernel method is because it leverages the spatial locality to improve the hit rate.

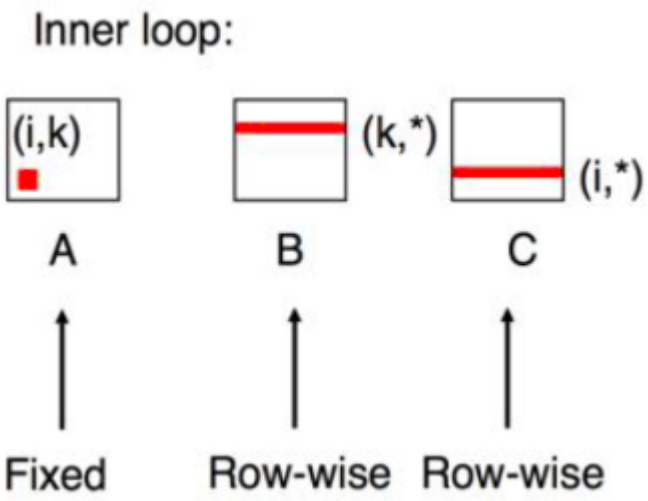
ijk algorithm




```
public static int[][] ijkAlgorithm(int[][] A, int[][] B) {
```

```
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}
```

ikj algorithm




```
public static int[][] ikjAlgorithm(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}
```

In both the algorithm, matrices are either accessed in row-major format or column major format. **Row major accesses tend to improve the perfomance as it supports spatial locality because of strided refrence patterns. whereas column-major memory accesses tend to consume more cycles, as it is a cache miss for every new column it access.**

Incase of ijk algorithm, the matrix A is accessed row-wise and B is accessed column-wise. Wheras in ikj algorithm, element of A is fixed that means accessed repeatedly (temporal locality) and B is accesed row-wise.

ikj alogrithm is cache friendly becuase it utilizes the benefits of both temporal and spatial locality to improve the hit rate. Thus ultimately make the multiplication faster than the prior one.

✓ 1m 2s completed at 1:21 PM