HPCA Assignment 1

Nikhil Tayade and **Pallavi Chauhan**
25952 and *26144*
nikhiltayade@iisc.ac.in and *pallavi2@iisc.ac.in*

System Configuration

- **Architecture**: x86_64
- **CPU op-mode(s)**: 32-bit, 64-bit
- **Address sizes**: 39 bits physical, 48 bits virtual
- **CPU(s)**: 12
- **On-line CPU(s) list**: 0-11
- **Model name**: 11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz
- **CPU family**: 6
- **Model**: 167
- **Thread(s) per core**: 2
- **Core(s) per socket**: 6
- **Socket(s)**: 1
- **Stepping**: 1
- **CPU MHz**: 800.000
- **CPU max MHz**: 4600.000
- **BogoMIPS**: 5424.00
- **L1d cache**: 288 KB
- **L1i cache**: 192 KB
- **L2 cache**: 1 MB
- **L3 cache**: 12 MB
- **OS**: Linux
- **Tools**: perf, likwid, python, VSCode, Jupyter

1 Constructing Roofline Model

1.1 Matrix-Matrix Multiplication

1.1.1 Introduction:

Matrix-Matrix multiplication of B x C where B and C are matrices of size [4096 x 4096], with each element of the matrix being a double. Executing and constructing Roofline Model for 4 different approaches i.e. Simple-[i,j,k] loop order, Simple-[k,i,j] loop order, Tiled-[i,j,k] loop order, Tiled-[k,i,j] loop order. Using performance monitoring counters, measuring FLOPS(Floating point operations per second) and memory bandwidth utilization across the memory hierarchy (DRAM, L3, L2, L1 caches) to construct Roofline model for each implementation variant.

1.1.2 Performance Measurement:

Performance characterization utilizes hardware performance monitoring counters to measure:

- Total floating-point operations executed
- Memory traffic at each hierarchy level (L1, L2, L3, DRAM)
- Cache hit/miss rates
- Execution time for FLOPS calculation

The Operational intensity for each memory level is calculated as:

$$I_{\text{level}} = \frac{TotalFLOPS}{TotalMemoryTraffic(BytesTransferred)} \tag{1}$$

1.1.3 Roofline Model Construction:

The Roofline models shows multiple bandwidth ceilings according to different memory hierarchy levels. Each implementation is characterized by multiple points on the plot, with coordinates:

- x-coordinate: Operational intensity
- y-coordinate: Achieved performance in GFLOPS

1.1.4 Calculating Peak GFLOPS:

Peak GFLOPS is calculated as the theoretical maximum number of floating-point operations a processor can perform per second.

$$GFLOPS_{peak} = Number\ of\ cores \times Clock\ frequency(GHz) \times FLOP\ per\ cycle\ per\ core$$

i.e.

$$GFLOPS_{peak} = N_{\text{cores}} \times f_{\text{GHz}} \times FLOP_{\text{per cycle per core}}$$

But since execution is performed on only 1 thread, automatically defaulted to one core.
FMA (Fused Multiply-Add),a single instruction that performs both a multiplication and an addition operation

$$a \times b + c$$

in one step with a single rounding, leading to improved accuracy, performance, and energy efficiency compared to performing the multiply and add as separate instructions, is being used we can say that the number of FLOPS is 2.
Therefore,

$$GFLOPS_{peak} = 1 \times 4.6_{\text{GHz}} \times 2$$

$$GFLOPS_{peak} = 9.2\ GFLOPS$$

1.1.5 Observed Performance Metrics:

For each Matrix-Matrix Multiplication implementation, the following data was observed by using perf commands running on one thread.

Metric	**Simple (i,j,k)**	**Simple (k,i,j)**	**Tiled (i,j,k)**	**Tiled (k,i,j)**
Total FLOP	1,37,43,89,53,477	1,37,43,89,53,477	1,37,43,89,53,477	1,37,43,89,53,477
Data movement	177743938767	8496762936	839672098	651957351
Runtime	324.05s	56.17s	45.14s	33.88s
Operational Intensity (x)	0.096	0.424	2.021	2.452
Performance (y)	20.460	3.052	26.351	4.069

For the System, the following Bandwidths were observed using likwid commands running on one thread.

Memory Level	**Bandwidth (GB/s)**
L1 Cache	275.237
L2 Cache	144.758
L3 Cache	69.443
DRAM	19.822

1.1.6 Roofline Model Visualization:

The following is the Roofline Model obtained after computing the observed performance metric and plotting accordingly.

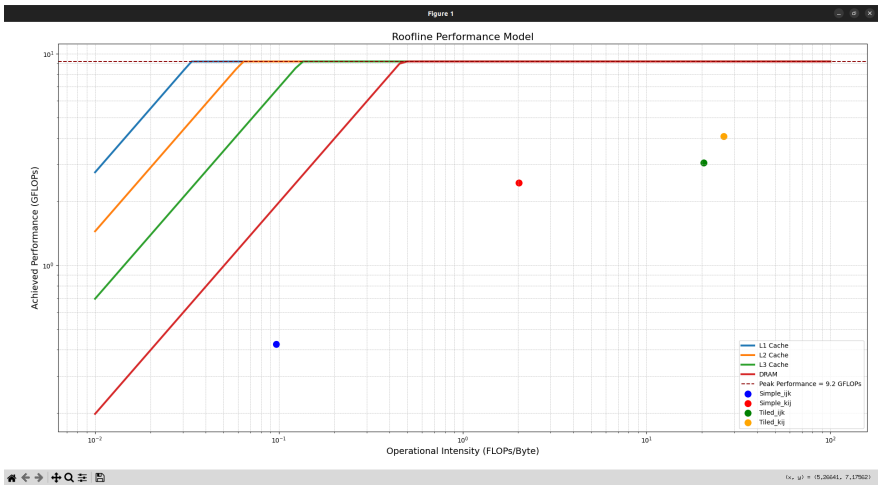

Figure 1: Roofline Model

1.1.7 Findings and Conclusions:

The progression clearly shows movement up the memory hierarchy:

1. Simple (i,j,k): DRAM-bound (excessive main memory access)
2. Simple (k,i,j): Compute-bound
3. Tiled implementations: Compute-bound

Loop Ordering Impact for Simple(i,j,k → k,i,j):

- Performance improvement: 0.424 → 2.447 GFLOPS (5.8x speedup)
- Runtime improvement: 324.05s → 56.17s (5.8x faster)
- Memory traffic reduction: 17.77 GB → 8.50 GB (2.1x less)
- Operational intensity increase: 0.096 → 0.424 FLOPS/Byte (4.4x higher)

The execution time improvements are substantial:

- Baseline: 5 minutes 24 seconds (Simple i,j,k)
- Loop optimization: 56 seconds (Simple k,i,j) – 5.8x faster
- Cache blocking: 45 seconds (Tiled i,j,k) – 7.2x faster
- Optimal combination: 34 seconds (Tiled k,i,j) – 9.6x faster

Combined optimization effect:

- Overall improvement: Simple (i,j,k) → Tiled (k,i,j) = 9.6x total speedup
- Operational intensity progression: 0.096 → 2.452 FLOPS/Byte (25.5x increase)
- Memory traffic reduction: 17.77 GB → 0.65 GB (27.3x less data movement)

The measured data validates the roofline model predictions:

- Simple implementations show memory bandwidth limitations
- Tiled implementations achieve compute-bound performance
- The progression follows expected roofline behavior with operational intensity improvements moving implementations from memory-bound to compute-bound regions

This analysis demonstrates that **systematic algorithmic optimization** through loop ordering can achieve nearly **10x performance improvement** while dramatically reducing memory traffic requirements.

1.1.8 Evidence:

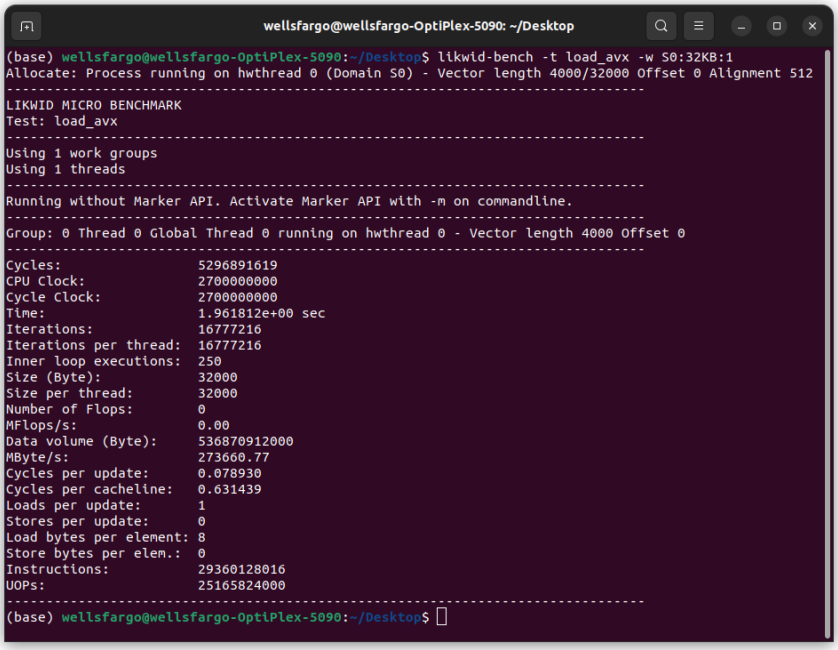

Figure 2: likwid command execution for L1 Bandwidth

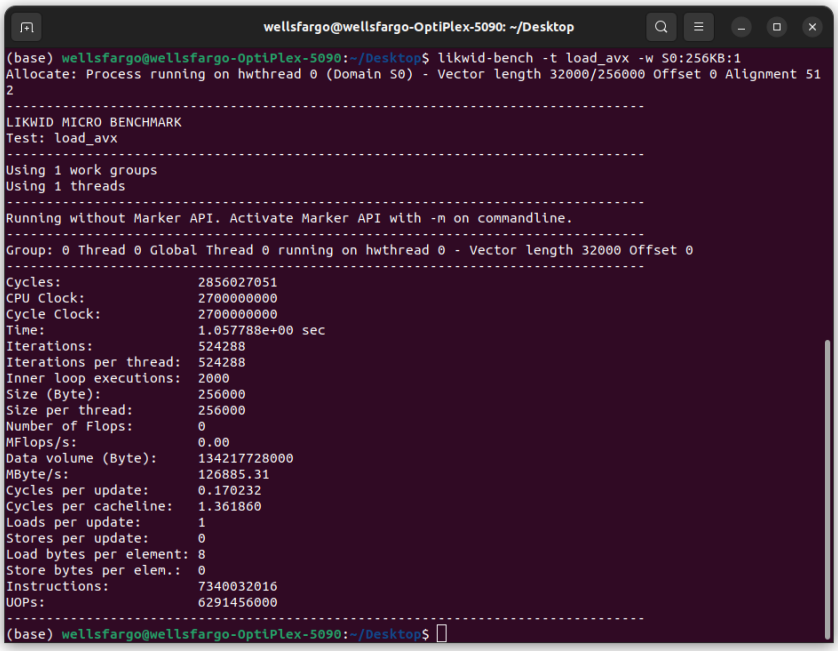

Figure 3: likwid command execution for L2 Bandwidth

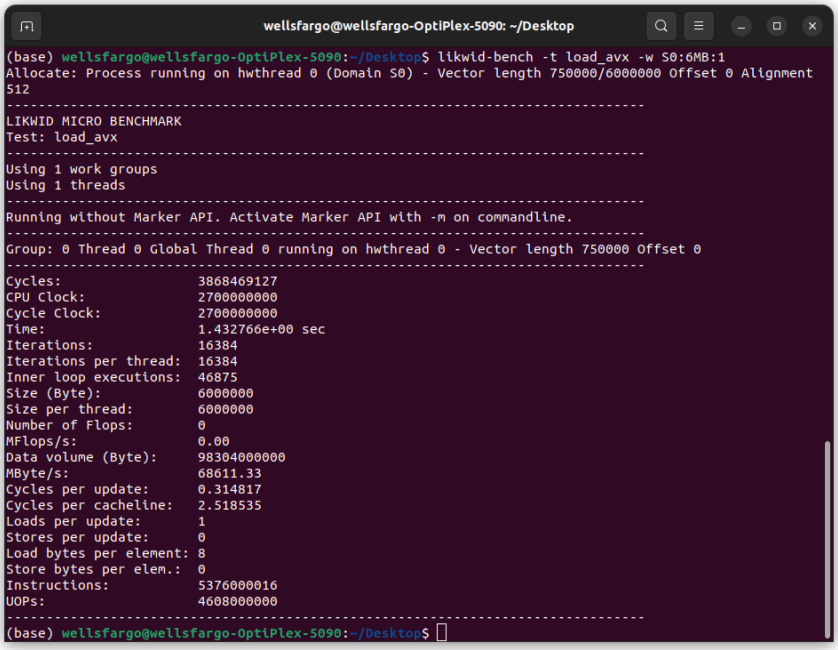

Figure 4: likwid command execution for L3 Bandwidth

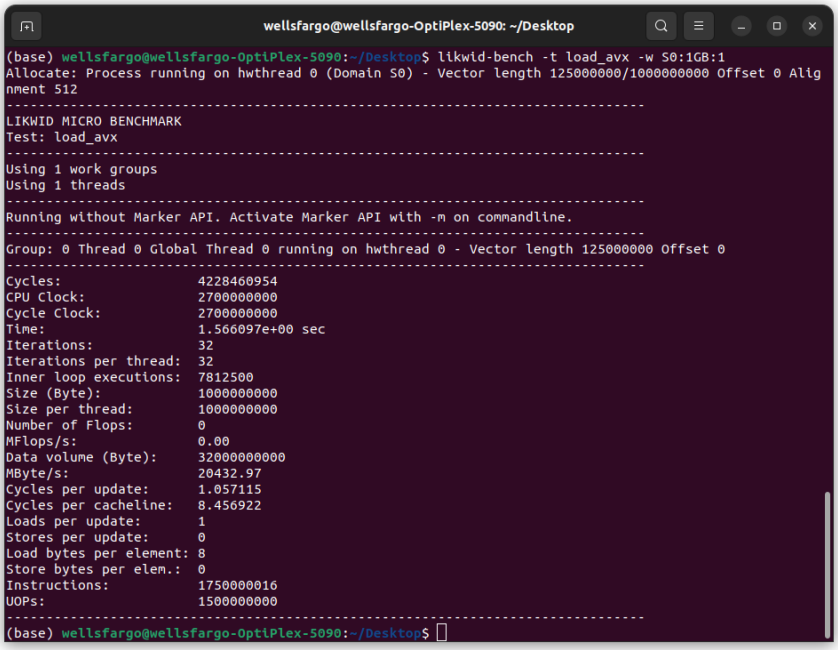

Figure 5: likwid command execution for DRAM Bandwidth

```
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ gcc -O2 -march=native -o matrix1op matrix1.c
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ taskset -c 0 perf stat -e cache-references,cache-misses,fp_arith_inst_retired.scalar_double,fp_arith_inst_retired.128b_packed_double
,fp_arith_inst_retired.256b_packed_double ./matrix1op
Allocating matrices...
Initializing matrices...
Running matmul_simple_ijk ...
Time (simple [i,j,k]): 323.947491 seconds

 Performance counter stats for './matrix1op':

 1,77,74,39,38,767      cache-references
    9,35,99,86,274      cache-misses                     #    5.27% of all cache refs
 1,37,43,89,53,477      fp_arith_inst_retired.scalar_double
                0      fp_arith_inst_retired.128b_packed_double
                0      fp_arith_inst_retired.256b_packed_double

     324.046135643 seconds time elapsed

     323.776178000 seconds user
       0.236973000 seconds sys
```

Figure 6: perf command execution for FLOP of Simple[ijk]

```
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ gcc -O2 -march=native -o matrix2op matrix2.c
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ taskset -c 0 perf stat -e cache-references,cache-misses,fp_arith_inst_retired.scalar_double,fp_arith_inst_retired.128b_packed_double
,fp_arith_inst_retired.256b_packed_double ./matrix2op
Allocating matrices...
Initializing matrices...
Running matmul_simple_kij ...
Time (simple [k,i,j]): 56.067523 seconds

 Performance counter stats for './matrix2op':

    8,49,67,62,936      cache-references
    7,93,50,96,044      cache-misses                     #   93.39% of all cache refs
 1,37,43,89,53,477      fp_arith_inst_retired.scalar_double
                0      fp_arith_inst_retired.128b_packed_double
                0      fp_arith_inst_retired.256b_packed_double

     56.172468487 seconds time elapsed

     56.062788000 seconds user
      0.106988000 seconds sys
```

Figure 7: perf command execution for FLOP of Simple[kij]

```
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ gcc -O2 -march=native -o matrix3op matrix3.c
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ taskset -c 0 perf stat -e cache-references,cache-misses,fp_arith_inst_retired.scalar_double,fp_arith_inst_retired.128b_packed_double
,fp_arith_inst_retired.256b_packed_double ./matrix3op
Allocating matrices...
Initializing matrices...
Running matmul_tiled_ijk ...
Time (tiled [i,j,k]): 45.035483 seconds

 Performance counter stats for './matrix3op':

     83,96,72,098      cache-references
     26,83,57,830      cache-misses                     #   31.96% of all cache refs
 1,37,43,89,53,477      fp_arith_inst_retired.scalar_double
                0      fp_arith_inst_retired.128b_packed_double
                0      fp_arith_inst_retired.256b_packed_double

     45.138438227 seconds time elapsed

     45.034506000 seconds user
      0.102989000 seconds sys
```

Figure 8: perf command execution for FLOP of Tiled[ijk]

```
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ gcc -O2 -march=native -o matrix4op matrix4.c
wellsfargo@wellsfargo-OptiPlex-5090:~/Desktop/ca1$ taskset -c 0 perf stat -e cache-references,cache-misses,fp_arith_inst_retired.scalar_double,fp_arith_inst_retired.128b_packed_double
,fp_arith_inst_retired.256b_packed_double ./matrix4op
Allocating matrices...
Initializing matrices...
Running matmul_tiled_kij ...
Time (tiled [k,i,j]): 33.782670 seconds

 Performance counter stats for './matrix4op':

     65,19,57,351      cache-references
     10,96,70,916      cache-misses                     #   16.82% of all cache refs
 1,37,43,89,53,477      fp_arith_inst_retired.scalar_double
                0      fp_arith_inst_retired.128b_packed_double
                0      fp_arith_inst_retired.256b_packed_double

     33.882118106 seconds time elapsed

     33.772383000 seconds user
      0.109988000 seconds sys
```

Figure 9: perf command execution for FLOP of Tiled[kij]

1.2 Breadth First Search program in the GAP benchmark suite

1.2.1 Introduction:

- Implementation: The GAP Benchmark Suite's BFS implementation targeting a large graph (e.g., 2^{25} vertices). Executed only the computation phase for measurements, excluding initialization.

- The total number of edges will be below as it follows Kronecker's graph with degree 15.

$$Edges\ Traversed\ =\ 523,609,147\ Edges$$

- Performance Metric: Since BFS has none floating point operations, Traversed Edges Per Second (TEPS) are used as the throughput metric. TEPS measures the number of edges explored per second, making it suitable for graph traversal tasks like BFS.

1.2.2 Performance Measurement:

- Since LLC misses typically go to main memory,

$$Total\ Bytes\ Loaded\ from\ Memory = from\ Memory\ MissesBytes\ from\ memory$$

$$= LLC\ misses \times 64bytes = 3,354,231,101 \times 64 = 214,670,790,464\ bytes = 214.67\ GB$$

- Execution of the BFS on the system on graph with 2^{25} vertices. Calculation of TEPS can be done as followed.

- The time taken for BFS Traversal is as follows:

$$BFS\ traversal\ time = 0.24258\ seconds$$

- Calculation:

$$TEPS = \frac{Total\ Traversed\ Edges}{Elapsed\ Time\ (seconds)}$$

$$TEPS = 2.158\ G\ Edges\ per\ sec$$

- So BFS performance is about 2.16 billion traversed edges per second.

- Max TEPS can be calculated by generating graphs with increasing number of vertices exponentially and executing BFS on it. Analyzing the number of edges traversed with the time obtained for that particular number of vertices will give us TEPS and the maximum TEPS will be obtained when the data is only retrieved from L1 cache i.e. only hits in L1 caches will exhibit that the TEPs traversed will be maximum.

$$Max\ TEP = 145\ GTEP$$

1.2.3 Roofline Model Construction:

- Y-Axis: For BFS, use TEPS instead of GFLOPS on the vertical axis, as it more accurately represents throughput for graph workloads.
- X-Axis (Operational Intensity): Define intensity as "edges traversed per byte moved" from main memory:

$$OperationalIntensity = \frac{Traversed\ Edges}{Bytes\ accessed\ from\ DRAM}$$

- Bytes from Memory = 214.67 GB
- Edges Traversed = 523,609,147
- So, the Operation Intensity will be

$$Operational\ Intensity = \frac{523,609,147}{214.67G}$$

- Therfore,

$$Operational\ Intensity = 0.0024$$

1.2.4 Roofline Model Visualization:

The following is the Roofline Model obtained after computing the observed performance metric and plotting accordingly.

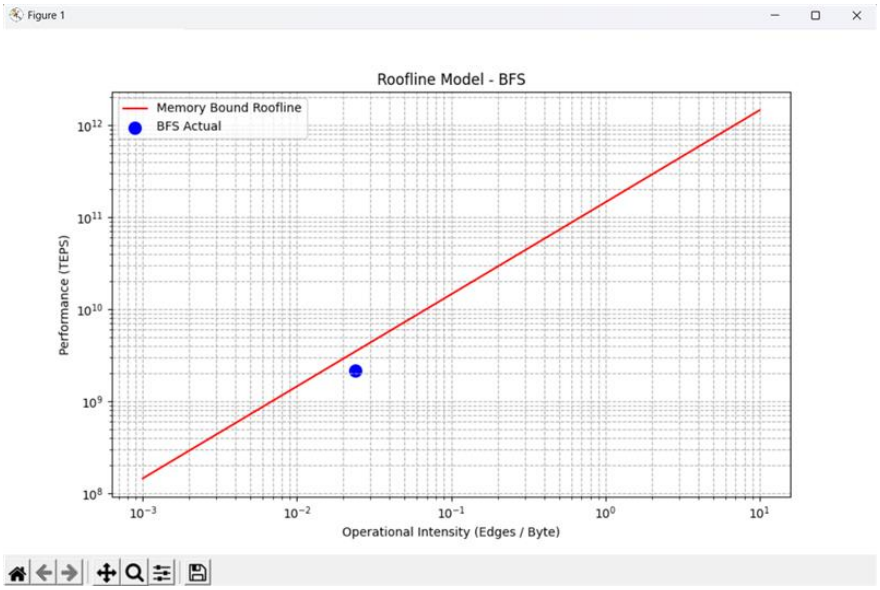

Figure 10: Roofline Model

1.2.5 Findings and Conclusion:

- As observed from the roofline plot, the GAPBS BFS is heavily memory bounded because the dominant cost is memory traffic, not computation. Main computation for BFS is finding the source which is heavily memory bounded by pointer chasing.

TEPS as a Performance Metric:

- Suitability: TEPS is meaningful for BFS/graph analytics because the dominant cost is memory traffic, not computation.
- Limitations: TEPS does not directly expose architectural details (e.g., cache effects) and may hide data which is different for other algorithms that traverse the same edges with different memory usage patterns.

1.2.6 Evidence:

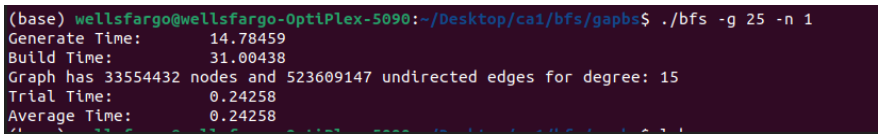

Figure 11: Command to execute GAPBS BFS with 2^{25} vertices

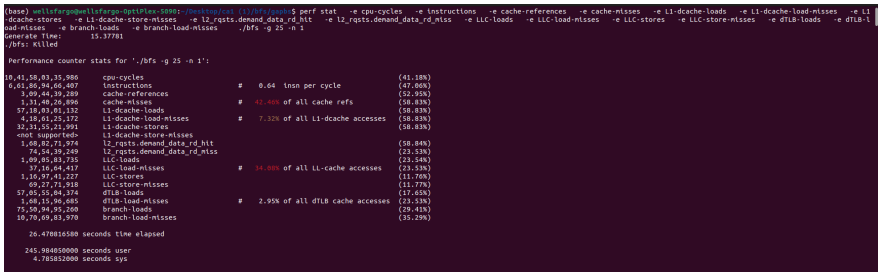

Figure 12: Command to execute GAPBS BFS with 2^{25} vertices and collect data with perf

2 CPI Stack for Programs

2.1 Introduction

Analyzing detailed performance of Breadth-First Search (BFS) algorithms from the GAP Benchmark Suite and Rodinia benchmark using IPC characteristics and CPI stack decomposition. The GAP BFS was tested on a graph with 2^{22} vertices, while NeoRodinia BFS was run on a graph with 20 million vertices.

2.2 IPC Characteristic Analysis

2.2.1 Approach:

- The BFS code was executed for both the Benchmark Suites i.e. GAPBS and Rodinia with performance monitoring tools (`perf`) to capture retired instructions and CPU cycle counts over short intervals of fixed instruction counts (e.g., 100M instructions per interval).

- IPC for each interval was calculated as:

$$\text{IPC} = \frac{\text{Retired Instructions}}{\text{CPU Cycles}}$$

- The full program execution was segmented into these intervals to analyze phase behavior over at least 10 billion instructions total.

2.2.2 Data Collection:

- Using the below command, the data for characterization of IPC is obtained via Hardware Performance Counters.

```
sudo perf stat -e L1-dcache-loads,L1-dcache-load-misses,
l2_rqsts.demand_data_rd_hit,l2_rqsts.demand_data_rd_miss,
LLC-load-misses,LLC-loads,dTLB-loads,dTLB-load-misses,
iTLB-load-misses,branch-instructions,branch-misses,cpu-cycles,
instructions -I 10 -o perf_raw.txt -- ./bfs -g 22 -n 20
```

Listing 1: perf command GAP Benchmark Suite

```
sudo perf stat -e L1-dcache-loads,L1-dcache-load-misses,
l2_rqsts.demand_data_rd_hit,l2_rqsts.demand_data_rd_miss,
LLC-load-misses,LLC-loads,dTLB-loads,dTLB-load-misses,
iTLB-load-misses,branch-instructions,branch-misses,
cpu-cycles,instructions -I 10 -o data2.txt --
./bfs_omp_CPU_P3_clang_O1_exec graph20M.txt
```

Listing 2: perf command Rodinia Benchmark

- The overall data is observed and stored in **perf_raw.txt** file for GAPBS and in **data2.txt** file for Rodinia.

- A Sample Interval Data:

time	counts	unit events
0.010058712	40,962,295	L1-dcache-loads
0.010058712	93,060	L1-dcache-load-misses
0.010058712	18,940	l2_rqsts.demand_data_rd_hit
0.010058712	8,803	l2_rqsts.demand_data_rd_miss
0.010058712	16,595	LLC-loads
0.010058712	15,540	LLC-load-misses
0.010058712	44,405,827	dTLB-loads
0.010058712	51	dTLB-load-misses
0.010058712	314	iTLB-load-misses
0.010058712	41,298,248	branch-instructions
0.010058712	8,161	branch-load-misses
0.010058712	43,928,753	cpu-cycles
0.010058712	181,794,335	instructions

- The total instruction count across all intervals was verified to surpass 10 billion, fulfilling the assignment's coverage criteria.

- Retrieved **perf_raw.txt** and **data2.txt** file needs to be cleaned up and converted into **perf_counter_intervals.csv** and **data2.csv** respectively file for better compilation of data.

2.2.3 Result:

GAP Benchmark Suite

- The following IPC Characteristic graph is obtained for GAP Benchmark Suite, after using the information perceived via **perf_counter_intervals.csv** file.

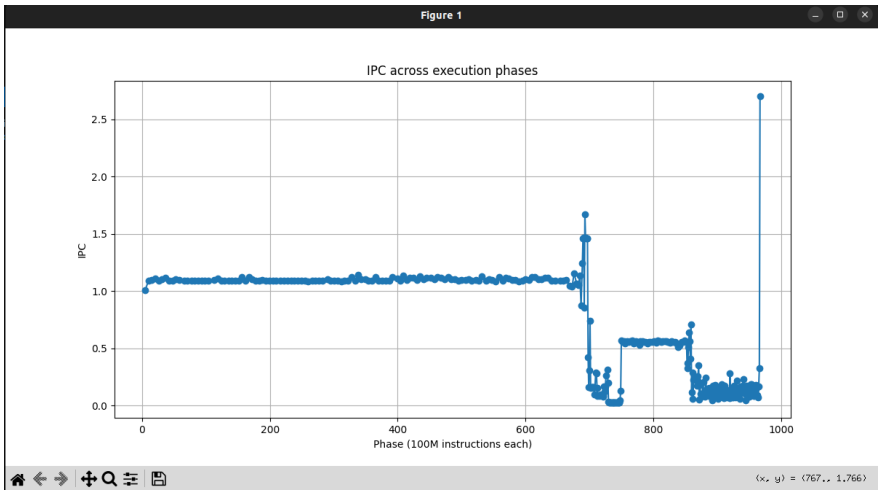

Figure 13: IPC Characteristic Plot for GAP Benchmark Suite

Rodinia Benchmark

- The following IPC Characteristic graph is obtained for Rodinia Benchmark, after using the information perceived via **data2.csv** file.

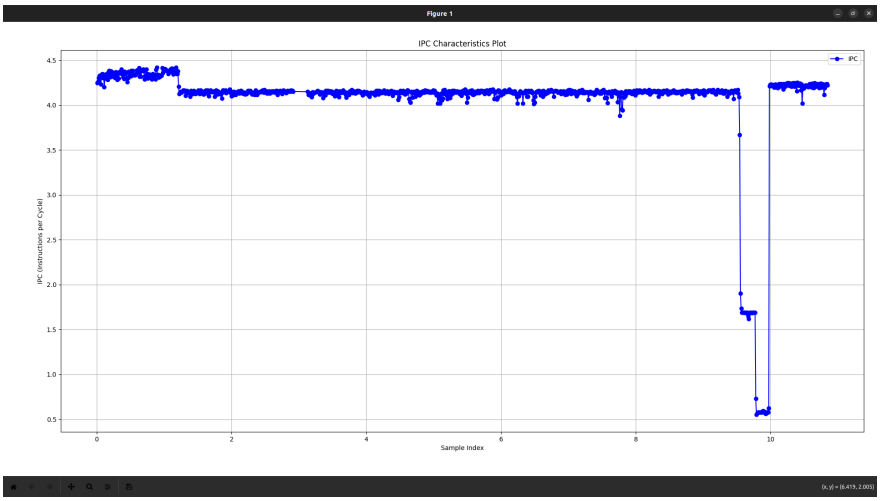

Figure 14: IPC Characteristic Plot for Rodinia Benchmark

2.2.4 Findings:

Phases and Transitions:

- Both the plots displays several distinct regions where the IPC value remains relatively stable, interrupted by sharp transitions. Each plateau or region suggests a different computational phase or workload characteristic of the BFS algorithm. The sharp transitions between plateaus are likely associated with shifts between graph setup, traversal or queue management.

GAP Benchmark Suite

- **Initialization and High IPC Regions:** The initial segment with higher IPC (> 1) most likely corresponds to the initialization phase, where the program is preparing data structures. This phase tends to be compute-bound.
- **Dips and Low IPC Regions:** Noticeable drops to IPC < 0.5 highlight memory-bound phases, typical of the main BFS traversal. Breadth-First Search, especially in large, irregular graphs, involves unpredictable pointer chasing and sparse memory accesses, Because of which IPC decreases.

Rodinia Benchmark

- **Initialization and High IPC Regions:** The initial segment with higher IPC ($4 < IPC < 4.5$) most likely corresponds to the initialization phase, where the program is preparing data structures. This phase tends to be compute-bound
- **Next Stable but High IPC Region:** Slightly lower IPC (just above 4) possibly represents the main BFS traversal process.

- **Dips and Low IPC Regions:** IPC falls drastically to below 1, showing a clear entry into a memory-bound.
- **Sharp Spikes:** IPC increases drastically, possibly exhibiting entry into clean-up phase

2.3 CPI Stack and Linear Regression Model

2.3.1 Approach:

The CPI stack constructed using hardware performance counters measures the counts of key microarchitectural events—L1 instruction cache misses, L1 data cache misses, L2/L3 cache misses, TLB misses and branch misses. A linear regression model with non-negative coefficients was fit to relate these miss events additively to total CPI.
The CPI Stack tells us which kinds of stalls or delays dominate the processor's cycle consumption relative to instruction execution, helping computer architects understand and optimize the underlying processor design or software behavior affecting performance.

2.3.2 Data Collection:

Using the same data obtained via command in Listing 1 and Listing 2 for GAP Benchmark Suite and Rodinia Benchmark respectively, CPI stack can be computed .

2.3.3 Linear Regression Model for CPI Stack:

- Feature Selection: Select miss-events

$$\{E_{\text{L1I}}, E_{\text{L1D}}, E_{\text{L2}}, E_{\text{L3}}, E_{\text{ITLB}}, E_{\text{DTLB}}, E_{\text{BRL}}, \dots\}.$$

- A multiple linear regression with additive components was developed where the total CPI is modeled as:

$$\text{CPI} = \beta_0 + \sum_i \beta_i \times \text{MissEvents}_i$$

- Constraints enforced all $\beta_i \geq 0$ to reflect positive latency contributions.
- The model was trained on hundreds to thousands of intervals per BFS benchmark.
- Model Evaluation:

Table 1: Regression Quality Metrics

	RMSE	R^2	Adj. R^2	F-statistic (p-value)
GAP BFS	0.00	1.00	0.00	1.6×10^{31}
Rodinia BFS	0.00	1.00	0.00	2.4×10^{29}

2.3.4 Result:

GAP Benchmark Suite

- The following CPI Stack is obtained for GAP Benchmark Suite, after using the information perceived via **perf_counter_intervals.csv** file.

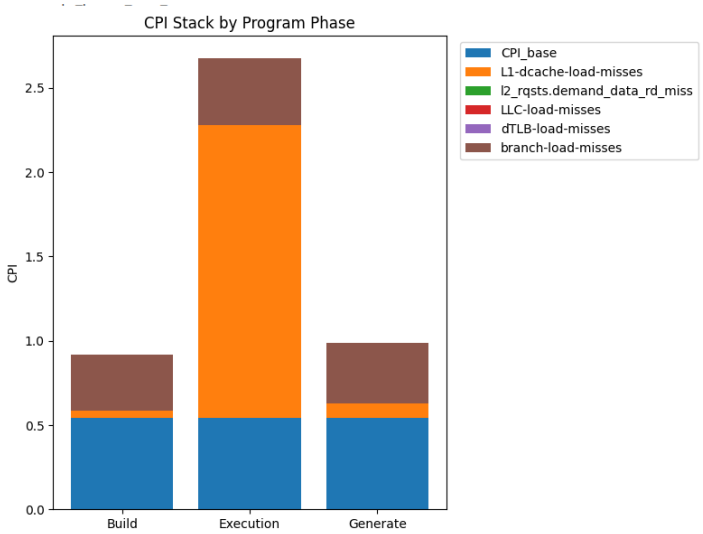

Figure 15: CPI Stack for GAP Benchmark Suite

Rodinia Benchmark

- The following CPI Stack is obtained for Rodinia Benchmark, after using the information perceived via **data2.csv** file.

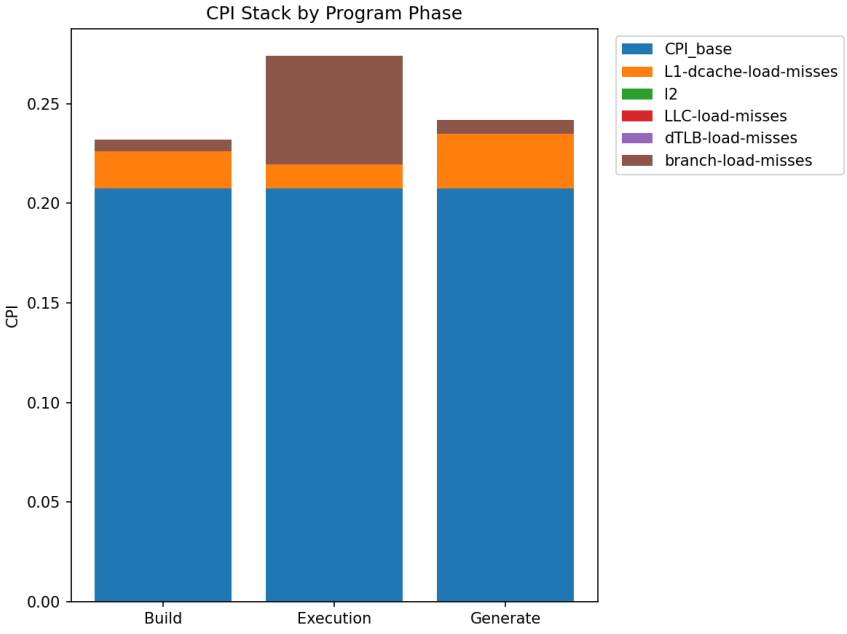

Figure 16: CPI Stack for Rodinia Benchmark

2.3.5 Findings:

GAP Benchmark Suite

- **Build Phase:** CPI is moderate and almost evenly split between base CPI and branch misses, with a lower but present L1 data cache miss contribution.
- **Execution Phase:** The single largest CPI contributor is L1-dcache-load-misses, which overshadows other components and pushes total CPI above 2.5. Branch-load-misses make a smaller but significant contribution.
- **Generate Phase:** CPI falls to a moderate value, with strong contributions from both branch-load-misses and base CPI, but a noticeably smaller L1 cache miss impact than in execution.

Rodinia Benchmark

- **Build Phase:** Dominated by base CPI, with a smaller, but visible, contribution from L1 data cache misses and minor branch miss effects.
- **Execution Phase:** Shows the highest overall CPI stack, with a substantial jump in branch-load-miss contribution. Minor addition from L1-dcache-load-misses, but the dominant bottleneck here is branch-load-misses.
- **Generate Phase:** Similar to the build phase, base CPI dominates, with some L1 cache and branch miss contributions.

References

[1] Scott Beamer, Krste Asanovic, and David Patterson, "The GAP Benchmark Suite," arXiv preprint arXiv:1508.03619, 2015.

[2] Scott Beamer, "GAP Benchmark Suite Reference Code," https://github.com/sbeamer/gapbs, v1.0, 2017.

[3] Pallavi Chauhan, "Program Code Data," https://github.com/pallavi-web/CA_Assignment1, v1.0, 2017.