HW3 PAPI Hardware Counters Study for CS553

Coen Petto

***Abstract*—This study is my first step into hardware counters using PAPI. Two different GEMM implementations will be compared against one another. I attempt to categorize the differences in program behavior from different counters as well as derive my own useful metrics from them.**

I. Introduction

THIS was performed on MAGGOT the CS machine with an Intel(R) Xeon(R) E-2278G CPU @ 3.40GHz and 8cores/16threads. My metrics were collected in the middle of the night, with no other users on the machine.

- This code was compiled with the POLYBENCH 4.2.1 harness, without RESTRICT, and with GCC 13.2 -O3.
- The counters were collected using PAPI version 5.6
- Two versions of GEMM are compared against each other. The standard GEMM loops (i, k, j) and the EVALLAB GEMM loops (i, j, k). The code is displayed below.

```
for (i = 0; i < _PB_NI; i++) {
   for (j = 0; j < _PB_NJ; j++)
      C[i][j] *= beta;
   for (k = 0; k < _PB_NK; k++) {
     for (j = 0; j < _PB_NJ; j++)
       C[i][j] += alpha * A[i][k] * B[k][j];
   }
 }
```

Listing 1. Standard GEMM

```
for (i = 0; i < _PB_NI; i++) {
   for (j = 0; j < _PB_NJ; j++)
   C[i][j] *= beta;
   for (j = 0; j < _PB_NJ; j++) {
     for (k = 0; k < _PB_NK; k++)
   C[i][j] += alpha * A[i][k] * B[k][j];
   }
 }
```

Listing 2. EVALLAB GEMM

II. Method

All experiments are driven by my `tuning2.py` script.

```
python3 tuning2.py \
  --profile-papi \
  --papi-counter-file myfile.txt \
  --read-environment openmp_env.txt \
  --explore-polybench <kernel>.c \
  --explore-N "256" \
=
```

which compiles each binary and launches 10 timed runs as well as one final run for PAPI counters. All needed information is generated into a CSV file which, will be broken down and displayed in a series of tables. The counters available on my machine are in the following list, and will be detailed if used for computed metrics.

A. Collected Hardware Counters

All collected counters for this report are:

- **Cycles & Stalls:**
  - `PAPI_TOT_CYC`
  - `PAPI_RES_STL`
- **L1 Data Cache:**
  - Standard PAPI: `PAPI_L1_DCM`, `PAPI_L1_ICM`, `PAPI_L1_TCM`, `PAPI_L1_LDM`, `PAPI_L1_STM`
  - Native events: `mem_load_retired.l1_miss`, `mem_load_retired.l1_hit` `mem_inst_retired.all_loads`, `mem_inst_retired.all_stores`
- **L2 Cache:**
  - Standard PAPI: `PAPI_L2_TCA`, `PAPI_L2_TCM`, `PAPI_L2_TCR`, `PAPI_L2_TCW`, `PAPI_L2_DCA`, `PAPI_L2_DCM`, `PAPI_L2_DCR`, `PAPI_L2_DCW`, `PAPI_L2_ICA`, `PAPI_L2_ICM`, `PAPI_L2_ICR`, `PAPI_L2_ICH`, `PAPI_L2_LDM`, `PAPI_L2_STM`
  - Native events: `L2_RQSTS:ALL_DEMAND_DATA_RD`, `L2_RQSTS:ALL_RFO`, `L2_RQSTS:DEMAND_DATA_RD_MISS`, `L2_RQSTS:RFO_MISS`
- **L3 Cache:**
  - `PAPI_L3_TCM`, `PAPI_L3_LDM`, `PAPI_L3_DCA`, `PAPI_L3_DCR`, `PAPI_L3_DCW`, `PAPI_L3_ICA`, `PAPI_L3_ICR`, `PAPI_L3_TCA`, `PAPI_L3_TCR`, `PAPI_L3_TCW`
- **TLB:**
  - `PAPI_TLB_DM`, `PAPI_TLB_IM`
- **Vector Instructions:**
  - `PAPI_VEC_SP`, `PAPI_VEC_DP`
- **Instruction Mix:**
  - `PAPI_TOT_INS`, `PAPI_LD_INS`, `PAPI_SR_INS`, `PAPI_LST_INS`
- **Floating-Point Ops:**
  - `PAPI_SP_OPS`, `PAPI_DP_OPS`

TABLE I
DERIVED PERFORMANCE METRICS AT $N = 256$

Implementation	IPC	vIPC	%L1m_proxy	%L2m_demand	L1AC	vL1AC	%Stall	Avg (s)
Standard GEMM	3.228	0.795	8.262	8.494	0.369	1.499	14.400	0.006
EVALLAB GEMM	1.368	0.454	33.909	88.412	0.333	1.002	53.690	0.024

TABLE II
ABSOLUTE VALUES FOR DERIVED PERFORMANCE METRICS AT $N = 256$

Implementation	IPC (cyc)	vIPC (cyc)	%L1m_proxy	%L2m_demand	L1AC (ins)	vL1AC (vec_ins)	%Stall (cyc)
Standard GEMM							
Numerator	68 553 696	16 875 588	2 087 840	150 019	25 297 004	25 297 004	3 057 365
Denominator	21 234 273	21 234 273	26 583 357	1 766 150	68 553 696	16 875 520	21 234 273
EvalLab GEMM							
Numerator	151 718 642	50 364 416	8 687 562	15 099 504	50 462 847	50 462 849	59 551 272
Denominator	110 925 483	110 925 483	25 250 911	17 079 644	151 718 642	50 397 184	110 925 483

III. RESULTS

A. Derived-Metric Formulations

$$M = \text{mem\_load}$$

Then the seven derived metrics are:

$$\text{IPC} = \frac{tot\_ins}{tot\_cycl}, \tag{1}$$

$$\text{vIPC} = \frac{vec\_sp + vec\_dp}{tot\_cycl}, \tag{2}$$

$$\%\text{L1m}_{\text{proxy}} = \frac{\text{M\_retired.l1\_miss}}{\text{M\_retired.l1\_hit} + \text{M\_retired.l1\_miss}} \tag{3}$$

$$\%\text{L2m}_{\text{demand}} = \frac{\text{PAPI\_L2\_LDM} + \text{PAPI\_L2\_STM}}{\text{PAPI\_L2\_DCR} + \text{PAPI\_L2\_DCW}} \tag{4}$$

$$\text{L1AC} = \frac{loads + stores}{tot\_ins}, \tag{5}$$

$$\text{vL1AC} = \frac{loads + stores}{vec\_ins}, \tag{6}$$

$$\%\text{Stall} = \frac{\text{PAPI\_RES\_STL}}{tot\_cycl}. \tag{7}$$

IV. CHATGPT'S ANALYSIS

Why is gemm.c faster than gemm.evallab.c?

The Standard GEMM implementation significantly outperforms the EvalLab GEMM due to multiple interacting hardware efficiency factors, not just fewer total cycles. First, instruction-level parallelism is far superior in Standard GEMM, as shown by the IPC (3.228 vs. 1.368) and vector IPC (0.795 vs. 0.454). This indicates that Standard GEMM keeps the processor pipelines much busier, retiring more instructions and vector instructions per cycle. Second, memory access patterns in Standard GEMM are considerably better. Its L1 cache miss rate is roughly 4x lower (8.2% vs. 33.9%), and its L2 miss rate is almost 10x lower (8.5% vs. 88.4%). These differences are massive: EvalLab GEMM overwhelms the memory hierarchy, causing frequent cache misses and forcing much slower accesses to higher memory levels, thus stalling execution. This is reinforced by the % Stall Cycles: EvalLab GEMM spends more than half its execution time stalled (53.69%), whereas Standard GEMM only stalls 14.40% of the time. Finally, although both access memory at a comparable rate per instruction (L1AC is 0.369 vs. 0.333), Standard GEMM has a better vector load/store density (vL1AC of 1.499 vs. 1.002), suggesting more efficient vectorized memory operations when they do occur.

Thus, better cache utilization, more effective vectorization, lower memory latency, and reduced stalling all combine to make gemm.c significantly faster than gemm.evallab.c. The EvalLab version's different loop ordering degrades temporal locality, leading to higher cache miss penalties that the processor cannot hide through pipelining or speculation.

A. Comments on Absolute L1 and L2 Percentage Ratios

1) L1 Miss Proxy (L1 misses / (L1 hits + L1 misses)):

- Standard GEMM accesses the L1 cache slightly more than EvalLab GEMM (26.5M vs. 25.2M total L1 accesses).
- Despite more accesses, Standard GEMM has approximately 4x fewer L1 misses (2.1M vs. 8.7M misses).
- This indicates significantly better spatial and temporal locality in Standard GEMM's memory access patterns.
- EvalLab GEMM suffers from many more cold or conflict misses, even with fewer total L1 accesses.

2) L2 Miss Demand ((L2 load misses + store misses) / (L2 load accesses + store accesses)):

- EvalLab GEMM generates approximately 15M L2 misses, compared to only 150K in Standard GEMM — a 100x difference.
- EvalLab GEMM makes about 10x more total L2 accesses than Standard GEMM (17M vs. 1.7M accesses).
- Almost every L2 access in EvalLab GEMM results in a miss, indicating extremely poor cache locality.
- Standard GEMM maintains a cache-efficient working set, while EvalLab GEMM's loop structure disrupts locality severely.

3) Critical Insight on Absolute Values:

- EvalLab GEMM's performance collapse is not only due to worse miss *rates*, but also due to generating dramatically more total cache traffic.
- EvalLab GEMM overwhelms the memory hierarchy, saturating memory bandwidth and causing high pipeline stalls.
- Standard GEMM keeps memory traffic compact and cache-friendly, leading to significantly fewer stalls and better throughput.

V. Coen's Analysis

- **Speed**: The standard GEMM performs roughly 4x faster with roughly half as many instructions. This time "speed-up" is also reflected in the total cycles proportionally as roughly 4x. Standard GEMM instructions are a larger proportion vectorized than EvalLab, and a cycle is only stalled a third of the amount of time compared to the EvalLab GEMM.
- **IPC and vIPC**: The standard GEMM completes a lot more scalar and vector instructions per cycle (IPC = 3.228 vs. 1.368, vIPC = 0.795 vs. 0.454). An IPC as high as 3 suggests that the CPU is operating very efficiently. Instructions are being decoded, issued, and executed with minimal stalling or pipeline waste. The much lower IPC in EvalLab GEMM hints at a bottleneck. A possible reason could be the loop order of evallab, which reduces the prefetchers' ability to fetch needed cache lines efficiently due to padding/staggered accesses to the matrices like we spoke of in class. This would cause frequent memory stalls, preventing the CPU from fully utilizing its pipelines.
- **L1 Misses**: Raw L1 cache misses in EvalLab GEMM are 4x higher (8.7M misses vs. 2.1M), despite having slightly fewer total L1 accesses. So not only is the miss rate way worse percentage-wise, but there are also way more misses in absolute numbers. This leads to cache lines being evicted and reloaded repeatedly, dragging performance down.
- **L2 Misses**: L2 behavior in EvalLab is even worse. EvalLab GEMM triggers around 15M L2 misses compared to only about 150K for standard GEMM. An absurd 100x difference. Not only is the miss rate almost 90%, but it's amplified by the fact that EvalLab generates way more total L2 traffic overall (17M accesses vs. 1.7M). This furthers the idea of horrific cache line management due to inability to maintain locality.
- **Pipeline Stalls**: All the above issues stack up. EvalLab GEMM spends over half of all cycles stalled (53.69%), compared to only 14.4% in Standard GEMM. The EvalLab code isn't just less efficient, it spends most of its time sitting idle, waiting for data to arrive, rather than doing work.

VI. Conclusion

The performance gap between Standard GEMM and EvalLab GEMM is obvious, and captured logically in the disparity of cache/hardware metrics. Standard GEMM benefits from a better metric in all categories, due to the EvalLab GEMM suffering from poor data locality. Memory access patterns matter, and the simple changes to loop order resulted in drastic inefficiencies that could be analyzed with PAPI.

VII. Command Reference and Sample Output

Command:

```
python3 tuning2.py --profile-papi
    --explore-polybench gemm.c --explore-N
    "32"
```

Example Output:

```
N=32 min=0.000009s max=0.000010s avg=0.000010s
PAPI_TOT_CYC : 32,117
PAPI_RES_STL : 1,476
PAPI_L1_DCM : 418
PAPI_L1_ICM : 78
PAPI_L1_TCM : 524
PAPI_L1_LDM : 85
PAPI_L1_STM : 7
mem_load_retired.l1_miss : 56
mem_load_retired.l1_hit : 33,885
mem_inst_retired.all_loads : 34,368
mem_inst_retired.all_stores : 16,938
l1d.replacement : 418
PAPI_L2_TCA : 343
PAPI_L2_TCM : 933
PAPI_L2_TCR : 224
PAPI_L2_TCW : 8
PAPI_L2_DCA : 184
PAPI_L2_DCM : 871
PAPI_L2_DCR : 90
PAPI_L2_DCW : 8
PAPI_L2_ICA : 88
PAPI_L2_ICM : 67
PAPI_L2_ICR : 136
PAPI_L2_ICH : 37
PAPI_L2_LDM : 19
PAPI_L2_STM : 3
L2_RQSTS:ALL_DEMAND_DATA_RD : 104
L2_RQSTS:ALL_RFO : 6
L2_RQSTS:DEMAND_DATA_RD_MISS : 27
L2_RQSTS:RFO_MISS : 2
PAPI_L3_DCA : 878
PAPI_VEC_SP : 0
PAPI_VEC_DP : 34,304
PAPI_TOT_INS : 102,396
PAPI_LD_INS : 34,368
PAPI_SR_INS : 16,938
PAPI_LST_INS : 51,317
PAPI_SP_OPS : 0
PAPI_DP_OPS : 67,584
IPC : 3.1882
vIPC : 1.0681
%L1m_proxy : 0.81%
%L1m_load_true : 0.16%
%L2m_demand_data : 22.45%
%L2m_load : 21.11%
%L2m_store : 37.50%
%L2m_instr : 49.26%
%L2m_total : 272.01%
L1AC : 0.5011
vL1AC : 1.4956
%Stall : 4.60%
```