ECE 786: Advanced Computer Architecture: Data Parallel Processors Project Report

- Shantanu Mukhopadhyay[200541793]

-

Task 1: Cache Efficiency Analysis

For analysis, the following benchmarks were executed and categorized as following:

benchmark name	kernel_name	kernel_launch_uid	IPC with no cache bypassing	IPC with cache bypassing	% change of comparing the IPC with/without cache bypassing	benchmark category
rodinia.BP	_Z22bpnn_layerforward_CUDAPfS_S_S_ii	1	670.1913	666.3648	-0.57	Cache Insensitive
rodinia.BP	_Z24bpnn_adjust_weights_cudaPfiS_iS_S_	2	424.712	192.2678	-54.73	Cache Friendly
Rodinia.HS	_Z14calculate_tempiPfS_S_iiiiffffff	1	701.3718	707.6299	0.89	Cache Insensitive
rodinia.LUD	_Z12lud_diagonalPfii	1	0.7026	0.7176	2.13	Cache Insensitive
rodinia.LUD	_Z13lud_perimeterPfii	2	9.2446	9.1103	-1.45	Cache Insensitive
rodinia.LUD	_Z12lud_internalPfii	3	501.2445	567.1572	13.15	**Cache Unfriendly**
rodinia.LUD	_Z12lud_diagonalPfii	4	0.7558	0.7742	2.43	Cache Insensitive
rodinia.LUD	_Z13lud_perimeterPfii	5	10.9464	11.8102	7.89	**Cache Unfriendly**
rodinia.LUD	_Z12lud_internalPfii	6	497.3745	574.7466	15.56	Cache Unfriendly
rodinia.LUD	_Z12lud_diagonalPfii	7	0.7558	0.7741	2.42	Cache Insensitive
rodinia.LUD	_Z13lud_perimeterPfii	8	10.1697	10.9718	7.89	**Cache Unfriendly**
rodinia.LUD	_Z12lud_internalPfii	9	473.0808	557.2787	17.80	**Cache Unfriendly**
rodinia.LUD	_Z12lud_diagonalPfii	10	0.7558	0.7741	2.42	Cache Insensitive
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	1	217.5687	167.9066	-22.83	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	2	206.0139	146.9099	-28.69	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	3	165.9271	112.0179	-32.49	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	4	76.2236	61.3361	-19.53	Cache Friendly

ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	5	21.3021	36.1667	69.78	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	6	22.5533	44.4395	97.04	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	7	46.5675	86.5094	85.77	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	8	354.4445	455.3303	28.46	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	9	473.1056	486.792	2.89	Cache Insensitive
ISPASS.LPS	_Z13GPU_laplace3diiiiPfS_	1	383.1095	408.8568	6.72	**Cache Unfriendly**
ISPASS.NQU	_Z24solve_nqueen_cuda_kerneliiPjS_S_S_i	1	30.4185	30.7699	1.16	Cache Insensitive

Assuming 6% range for a benchmark to be Cache Insensitive.

Cache Unfriendly Benchmark list: **BFS** , **LPS and LUD** .

NOTE: all values are stored in Task 1 folder.

Task 2: Profiling-based Cache Bypassing:

Overview:

Must run the benchmark twice:

Step 1:For the first run of the simulation, I profiled the reference frequency of addresses in the data cache within each SM using a profiler and saved the reference counter in *profile.dump* in the following format:

< SM 0, kernel 1 : addr1 ref, addr2 ref, ...addrn ref kernel 2 : addr1 ref, addr2 ref, ...addrn ref

...

kernel n : addr1 ref, addr2 ref, ...addrn ref >

...

< SM n, kernel 1 : addr1 ref, addr2 ref, ...addrn ref kernel 2 : addr1 ref, addr2 ref, ...addrn ref

...

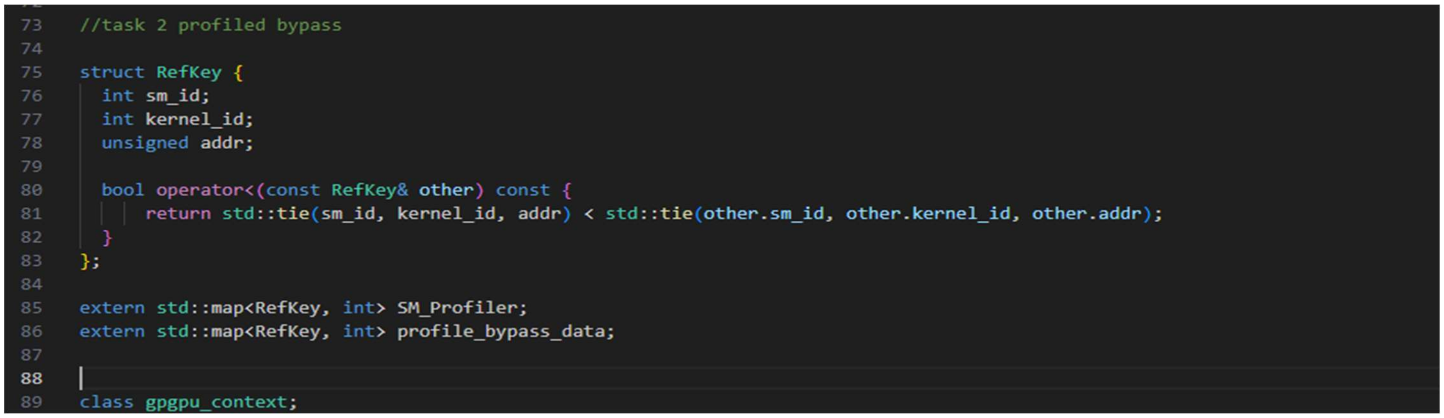
kernel n : addr1 ref, addr2 ref, ...addrn ref >

Step 2:
Read the profile.dump and I check if there are any addresses with references less than 3 and bypassed them.

Implementation:

Step1: Profiling and dumping stats to profile.dump

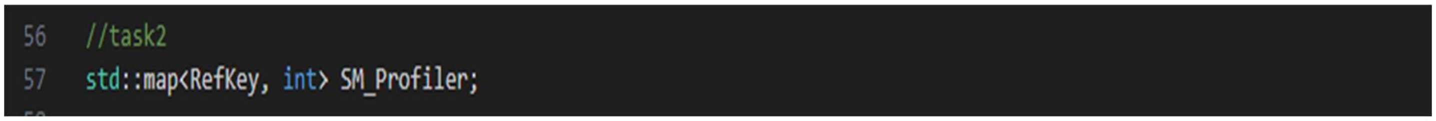
Changes in *gpgpu-sim/shader.h* file:



```
//task 2 profiled bypass

struct RefKey {
  int sm_id;
  int kernel_id;
  unsigned addr;

  bool operator<(const RefKey& other) const {
      return std::tie(sm_id, kernel_id, addr) < std::tie(other.sm_id, other.kernel_id, other.addr);
  }
};

extern std::map<RefKey, int> SM_Profiler;
extern std::map<RefKey, int> profile_bypass_data;


class gpgpu_context;
```

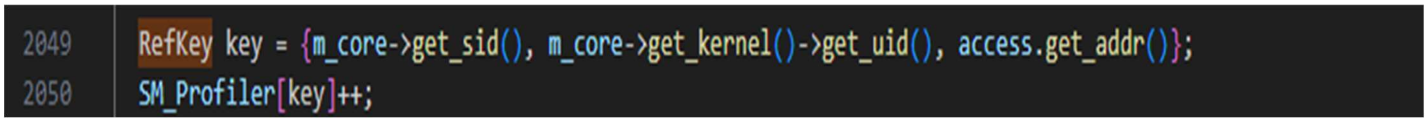
created a struct and tuple to map sm_id ,kenel_id,and addr inside a bool operator to keep maintain the order of References per SM.
These mappings were used in a hash map , so two hash maps used SM_profiler to actually get the values of the SM , kernel and addr and references data values and hash map profile_bypass_data for reading profile.dump of individual benchmarks and using this map, implemented the bypass logic.

Changes in *gpgpu-sim/shader.cc*



```
//task2
std::map<RefKey, int> SM_Profiler;
```

Inside memory_cycle()

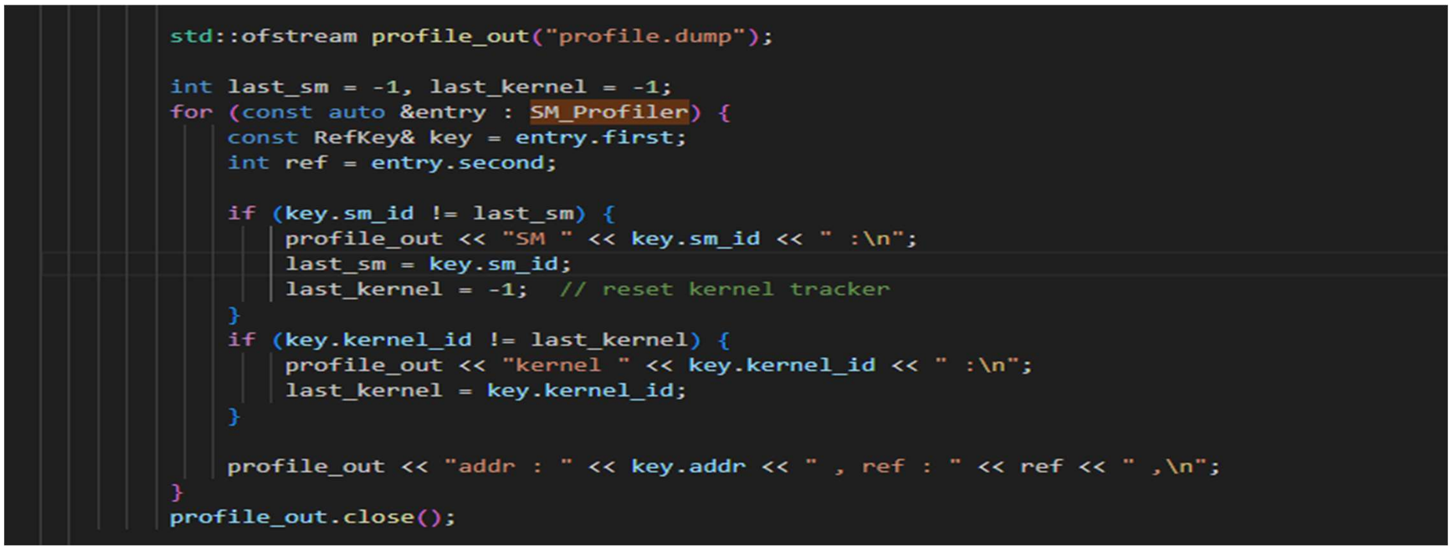


```
RefKey key = {m_core->get_sid(), m_core->get_kernel()->get_uid(), access.get_addr()};
SM_Profiler[key]++;
```

m_core()->get_sid() stores the SM value and get_kernel()->get_uid() stores the Kernel id inside the SM value and get_addr()- retrieves the addr which is accessing the L1D cache and are stored in Key , reference counter for each address encountered is then incremented and saved SM_Profiler hash map.

Changes in *gpgpusim_entrypointpoint.cc*

Inside gpgpu_context::print_simulation_time()



```
std::ofstream profile_out("profile.dump");

int last_sm = -1, last_kernel = -1;
for (const auto &entry : SM_Profiler) {
    const RefKey& key = entry.first;
    int ref = entry.second;

    if (key.sm_id != last_sm) {
        profile_out << "SM " << key.sm_id << " :\n";
        last_sm = key.sm_id;
        last_kernel = -1;  // reset kernel tracker
    }
    if (key.kernel_id != last_kernel) {
        profile_out << "kernel " << key.kernel_id << " :\n";
        last_kernel = key.kernel_id;
    }

    profile_out << "addr : " << key.addr << " , ref : " << ref << " ,\n";
}
profile_out.close();
```

Used standard C++ IO stream, set a default values for last_sm value, last_kernel for end of operation.

Read each entry in SM_Profiler hash map, were key is the first entry -which contains SM ,Kernel and addr information and the second information for the key is the reference counter of the address, and are printed in the above mentioned order in profile.dump file.

Commands to run the Benchmarks:

For BFS: ./ispass-2009-BFS graph65536.txt

For LPS: ./ispass-2009-LPS

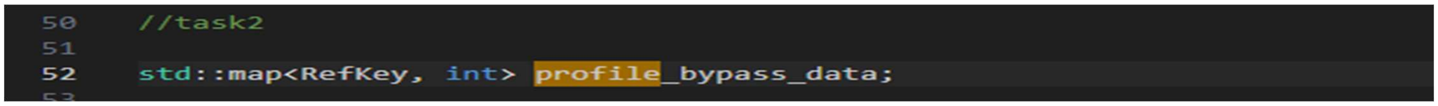
For LUD: ./lud-rodinia-3.1 -s 256 -v

After this step in each benchmark a profile.dump would be created indicating the SM, Kernel , addr and their reference counter.

Step 2: Profile based Bypass

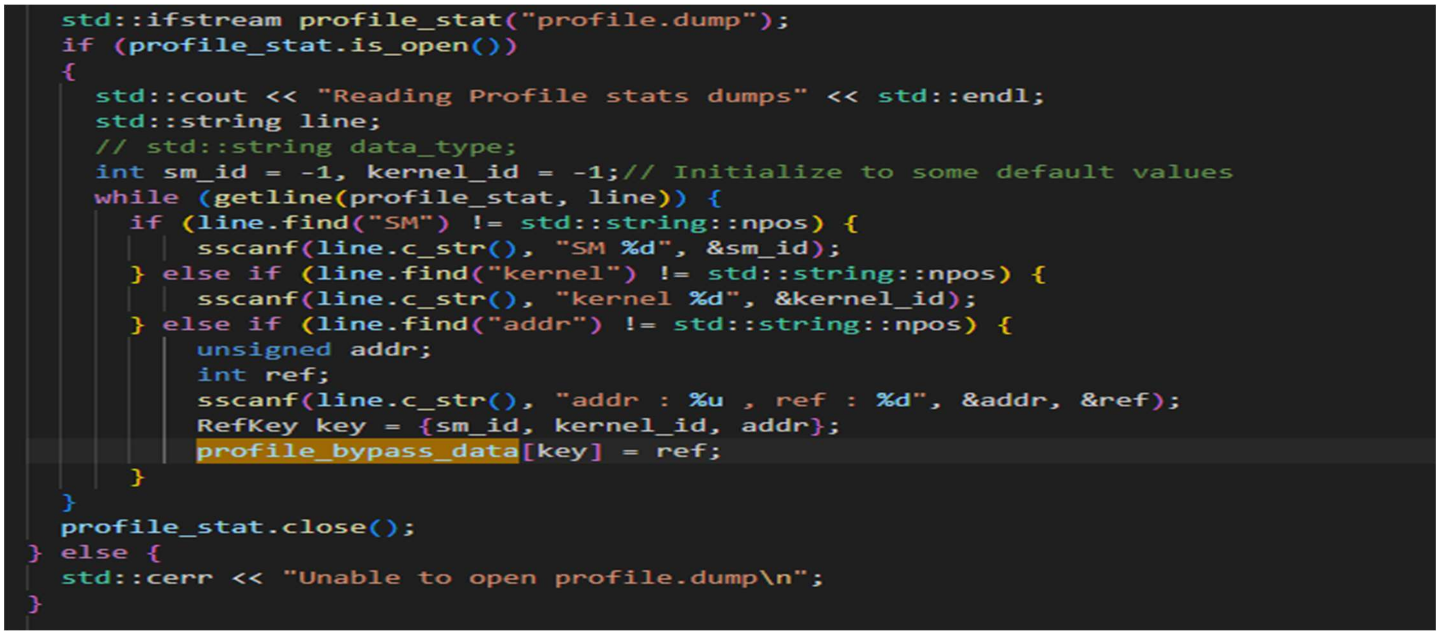
Since the benchmarks were run once to get the profile data dumps, we can use that file to read and use that such that if a reference counter for a addr is less than 3, we just bypass cache.

Changes in *gpgpusim_entrypointpoint.cc*



```
//task2

std::map<RefKey, int> profile_bypass_data;
```

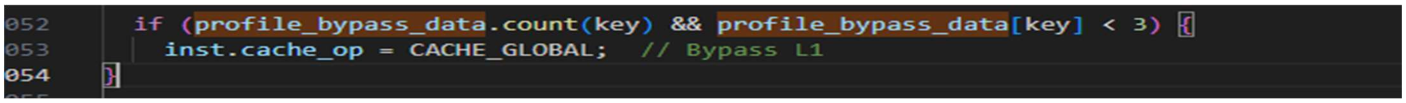
Inside gpgpu_sim *gpgpusim_context::gpgpu_ptx_sim_init_perf()



```
  std::ifstream profile_stat("profile.dump");
  if (profile_stat.is_open())
  {
    std::cout << "Reading Profile stats dumps" << std::endl;
    std::string line;
    // std::string data_type;
    int sm_id = -1, kernel_id = -1;// Initialize to some default values
    while (getline(profile_stat, line)) {
      if (line.find("SM") != std::string::npos) {
          sscanf(line.c_str(), "SM %d", &sm_id);
      } else if (line.find("kernel") != std::string::npos) {
          sscanf(line.c_str(), "kernel %d", &kernel_id);
      } else if (line.find("addr") != std::string::npos) {
          unsigned addr;
          int ref;
          sscanf(line.c_str(), "addr : %u , ref : %d", &addr, &ref);
          RefKey key = {sm_id, kernel_id, addr};
          profile_bypass_data[key] = ref;
      }
    }
    profile_stat.close();
} else {
    std::cerr << "Unable to open profile.dump\n";
}
```

Again I am using C++ standard input steam, to read the profile.dump file, iterating over each line in file if Line contains SM , kernel and addr we extract them and store them to Refkey struct and this struct along with an int is set to profile_bypass_data.

Changes in gpgpu-sim/shader.cc



```
    if (profile_bypass_data.count(key) && profile_bypass_data[key] < 3) {
      inst.cache_op = CACHE_GLOBAL;  // Bypass L1
    }
```

Checked the current memory access -Refkey key exists in the profiling data map or not and if the key exists checks if the reference count for that address is less than 3 we bypass the L1D cache by change the access instructions' cache_op to CACHE_GLOBAL.

Then we run the benchmarks again(reads the profile.dump and implements the bypass)

For BFS: ./ispass-2009-BFS graph65536.txt

For LPS: ./ispass-2009-LPS

For LUD: ./lud-rodinia-3.1 -s 256 -v

NOTE: All the profile.dump files for each benchmark are stored in Task 2/Profiled data

All output files with and without bypass are stored in Task 2/ Output Files

Output files with bypass are named *benchmark*_profile.txt

and output files without bypass are named *benchmark*.txt

and the modified files are stored in Task2/Modified Files

Results:

Run on BFS and LPS and LUD(cache unfriendly) on SM7_QV100 configuration:

Benchmark	Kernel	IPC without Profiled bypass	IPC with Profile Bypass	% change in IPC
BFS	5	87.2392	87.7643	0.601908
BFS	6	86.6631	83.9124	-3.17402
BFS	7	145.4857	133.107	-8.50853
BFS	8	229.1067	201.9134	-11.8693
LPS	1	638.116	667.4357	4.594729
LUD	3	712.5299	721.2603	1.225268
LUD	5	13.6126	14.3418	5.356802
LUD	8	12.6459	13.3236	5.359049
LUD	9	556.1233	557.6649	0.277205

For BFS: we see mostly that the IPC after profile based bypass has rather decreased a bit except in kernel 5 where we see minute increase in IPC.

For LPS and LUD: We can see some considerable increase in IPC for the kernels which were cache unfriendly as we saw in our previous run in task1 which is good improvement.

Task 3: Experiment with different cache replacement policies

To execute this change in gpgusim.config we change LRU (L) to FIFO(F) in the L1D cache.

-gpgpu_cache:dl1 N:32:128:4,**L**:L:m:N:H,S:64:8,8

To

-gpgpu_cache:dl1 N:32:128:4,**F**:L:m:N:H,S:64:8,8

benchmark name	**kernel_name**	**kernel_launch_uid**	**IPC with LRU**	**IPC with FIFO**	**percentage change of comparing the IPC of LRU vs FIFO**	**benchmark category**
rodinia.LUD	_Z12lud_diagonalPfii	1	0.7026	0.7026	0.00	Cache Insensitive
rodinia.LUD	_Z13lud_perimeterPfii	2	9.2446	9.2446	0.00	Cache Insensitive
rodinia.LUD	_Z12lud_internalPfii	3	501.2445	496.1378	-1.02	**Cache Unfriendly**
rodinia.LUD	_Z12lud_diagonalPfii	4	0.7558	0.7558	0.00	Cache Insensitive
rodinia.LUD	_Z13lud_perimeterPfii	5	10.9464	10.9464	0.00	**Cache Unfriendly**

rodinia.LUD	_Z12lud_internalPfii	6	497.3745	498.8101	0.29	Cache Unfriendly
rodinia.LUD	_Z12lud_diagonalPfii	7	0.7558	0.7558	0.00	Cache Insensitive
rodinia.LUD	_Z13lud_perimeterPfii	8	10.1697	10.1697	0.00	**Cache Unfriendly**
rodinia.LUD	_Z12lud_internalPfii	9	473.0808	463.917	-1.94	**Cache Unfriendly**
rodinia.LUD	_Z12lud_diagonalPfii	10	0.7558	0.7558	0.00	Cache Insensitive
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	1	217.5687	217.5687	0.00	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	2	206.0139	206.0139	0.00	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	3	165.9271	165.9271	0.00	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	4	76.2236	74.2745	-2.56	Cache Friendly
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	5	21.3021	20.9452	-1.68	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	6	22.5533	22.6421	0.39	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	7	46.5675	47.1472	1.24	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	8	354.4445	369.4767	4.24	**Cache Unfriendly**
ISPASS.BFS	_Z6KernelP4NodePiPbS2_S1_S2_i	9	473.1056	473.6455	0.11	Cache Insensitive
rodinia.BP	_Z22bpnn_layerforward_CUDAPfS_S_S_ii	1	670.1913	670.1913	0.00	Cache Insensitive
rodinia.BP	_Z24bpnn_adjust_weights_cudaPfiS_iS_S_	2	424.712	421.7699	-0.69	Cache Friendly

While comparing LRU vs FIFO replacement policy, we see that amongst the benchmarks run in Task 1 with config SM2_GX480(also used in this task), only few kernels are sensitive to FIFO replacement policy and the dip and rise of those kernel's IPC were small.

Some secondary stats to explain this could be stated below:

Benchmark	**kernel id**	**Rep Policy**	**L1D Total Accesses**	**L1D Miss Rate**	**read_global**	**write_global**	**load_insn**	**store_insn**	**IPC**
rodinia.LUD	3	LRU	15616	0.5963	9204	4080	184576	65280	501.2445
rodinia.LUD	3	FIFO	15616	0.5915	9145	4080	184576	65280	496.1378
rodinia.LUD	9	LRU	41171	0.6228	25308	10787	486144	172592	473.0808

rodinia.LUD	9	FIFO	41171	0.6223	25241	10787	486144	172592	463.917
ISPASS.BFS	4	LRU	53460	0.5371	12428	16750	293207	18227	76.2236
ISPASS.BFS	4	FIFO	54192	0.5506	13288	17026	293903	18639	74.2745
ISPASS.BFS	5	LRU	516193	0.8424	229538	206662	873432	303266	21.3021
ISPASS.BFS	5	FIFO	520218	0.844	232308	208127	878528	305700	20.9452
ISPASS.BFS	7	LRU	1216761	0.8586	653484	393846	1929022	647948	46.5675
ISPASS.BFS	7	FIFO	1213958	0.8593	653076	392769	1926555	646461	47.1472
ISPASS.BFS	8	LRU	1219223	0.8575	654203	393947	1994871	648049	354.4445
ISPASS.BFS	8	FIFO	1216500	0.8582	654203	392880	1992474	646574	369.4767

rodinia.LUD – Kernel 3: IPC **drops with FIFO:** Despite stable stats, temporal locality degradation due to FIFO caused minor IPC dip.

rodinia.LUD – Kernel 9: IPC drops IPC drop due to loss of effective reuse under FIFO for critical data even though quantitative stats don't diverge much.

ISPASS.BFS – Kernel 4: IPC drops Memory pressure increase under FIFO → longer memory latency exposure → IPC dip.

ISPASS.BFS – Kernel 5: IPC **drops** Even slight memory traffic increase → reduced warp throughput → IPC down.

ISPASS.BFS – Kernel 7: IPC rises Reduced memory ops despite higher miss rate = more active warps, better IPC.

ISPASS.BFS – Kernel 8: IPC rises Store pressure relief → memory latency reduction → more warps execute → IPC boost

On average, LRU performs better than FIFO, as it exploits temporal locality more effectively. While FIFO simply evicts the oldest entry regardless of usage, LRU evicts the least recently accessed entry, which is more likely to be truly unused. This leads to fewer unnecessary evictions of frequently accessed data, resulting in lower miss rates and improved IPC in many workloads. Although LRU may introduce slight hardware overhead due to tracking recent usage, the performance benefits often justify its use, especially in cache-sensitive applications, since it better uses temporal locality.

NOTE: All the files for FIFO execution are present in Task 3/*Benchmark* folder and to compare with LRU execution we can use the files generated in task1