Report:
Memory Coalescing for GPUs
Assignment 5 GPU Architecture Course
Ramyad Hadidi - November 20, 2015

---

1 Introduction

In this assignment, trace generation is implemented in previously developed Harmonica ISA emulator [1]. Also, in emulator different mechanisms for memory coalescing is integrated. Therefore, with the help of the emulator a study on memory coalescing effects on cache has been done using Dinero IV[2] a Trace-Driven Uniprocessor Cache Simulator. In this report first we show how memory coalescing code was integrated, then a study about impact of cache line size on cache performance, and finally another study about memory mapping schemes and its impact on cache performance.

The benchmarks that we are using is *small* version of *vecsum* and *scan* applications with 8 threads/warp and 32 threads/warp[3]. For Dinero, we use one level cache with fixed properties except of cache line size in the first study. We have used a 32KB, 4 way set-associative cache with 8 banks.

2 Developed Code

In each execution step, the main class of *core_c* will call *step()* function, which it will call all warp in the core and execute them. In each warp after executing all threads, if threads encounter ST or LD, they will save their address to variable *m_coalMemAddr* and the number of them in *m_uniqeCoalMemAddr*. At the end of the execution of a warp, a *coalesce()* function will be called and address will be coalesced based on global configurations, such as cache line size and mapping scheme. Here is the code:

```
1  void warp_c::coalesce() {
2    unsigned maximumDistance = (1LL << CACHE_LINE_IN_BYTE/
         WORD_SIZE_IN_BYTE) - 1;
3    set<Addr> memAddresses;
4    set<Addr> coalMemAddr;
5
6    //Get inputs
7    for (unsigned int i=0; i<m_uniqeCoalMemAddr ; i++)
8      memAddresses.insert( m_coalMemAddr[i] );
9
10   //Note: set will not insert duplicate element
11   for (set<Addr>::iterator itA = memAddresses.begin(); itA
         != memAddresses.end();) {
12     bool coalFound = false;
```

[1] https://github.com/ramyadhadidi/harmonica_emulator_proj
[2] http://pages.cs.wisc.edu/~markhill/DineroIV/
[3] 16 threads/warp was not supplied for small version of traces

```
13      for (set<Addr>::iterator itB = memAddresses.begin(); itB
            != memAddresses.end();) {
14        Addr firstElem = *itA >> RIGHT_BIT_SHIFT_COAL;
15        Addr secondElem = *itB >> RIGHT_BIT_SHIFT_COAL;
16        Addr mask = (1LL << RIGHT_BIT_SHIFT_COAL) - 1;
17        Addr lowerFirstElem = *itA & mask;
18        Addr lowerSecondElem = *itB & mask;
19        if ( (abs(secondElem - firstElem) < maximumDistance)
              && (*itB != *itA) && (lowerFirstElem ==
              lowerSecondElem) ) {
20          coalFound = true;
21          memAddresses.erase(itB++);
22          coalMemAddr.insert(*itA);
23        }
24        else {
25          itB++;
26                  m_core->statNotCoalesced ++;
27        }
28      }
29      //Uniqe item itslef, insert it
30      if (!coalFound)
31        coalMemAddr.insert(*itA);
32      memAddresses.erase(itA++);
33    }
34
35    //Restore outputs
36    int sizeOld = m_uniqeCoalMemAddr;
37    m_uniqeCoalMemAddr = coalMemAddr.size();
38    unsigned int i=0;
39    for (set<Addr>::iterator it = coalMemAddr.begin(); it !=
          coalMemAddr.end(); ++it, ++i)
40      m_coalMemAddr[i] = *it;
41    m_core->statCoalesced += sizeOld-m_uniqeCoalMemAddr;
42  }
```

First, the elements of m_coalMemAddr have been copied to a set. Set is used because it will not allow duplicate elements. Also, set will order the addresses which enable us to write a 2 for loop for coalescing and select the first item as the representative cache line if coalescing criteria passed. In the 2 for loop body we check if the 2 addresses has coalescing ability based on maximum distance (based on cache line size) and also if coalescing is based on higher value bits we check equality of lower bits. In addition I have used a statistics to show how many coalescing is occurring versus not coalesced accesses.

3 Cache Line Size Study

In this study coalescing based on lower bits is done and the impact of changing cache line size on different cache performance metrics has been studied. In figure 1 miss rate is shown for different warp size and cache line size. As we can see in all cases miss rate is decreasing. Also, an interesting point is between 8 warp and 32 warp version which shows 32 warp size is getting more benefit from cache line size 64 to 128 than 8 warp size. This is because 32 warp accesses has more covering. Figure 2 shows reduction in demand fetches and therefore pressure on memory system.

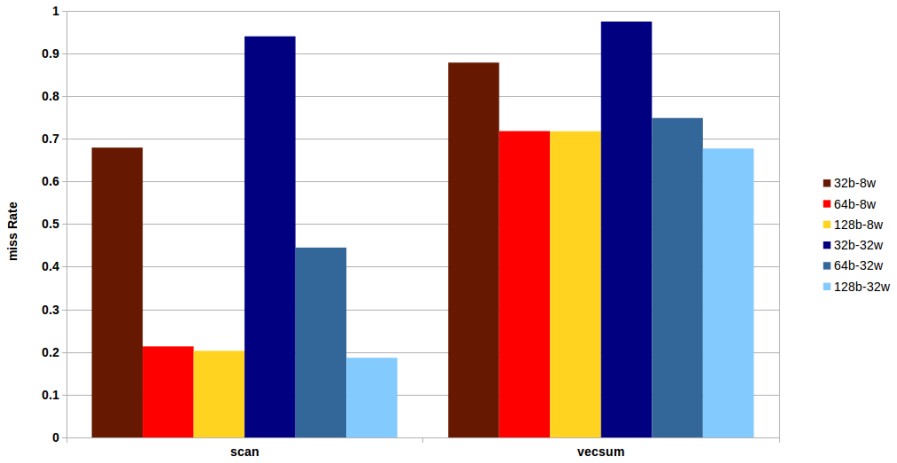

Figure 1: Miss Rate with Different Cache Line Size

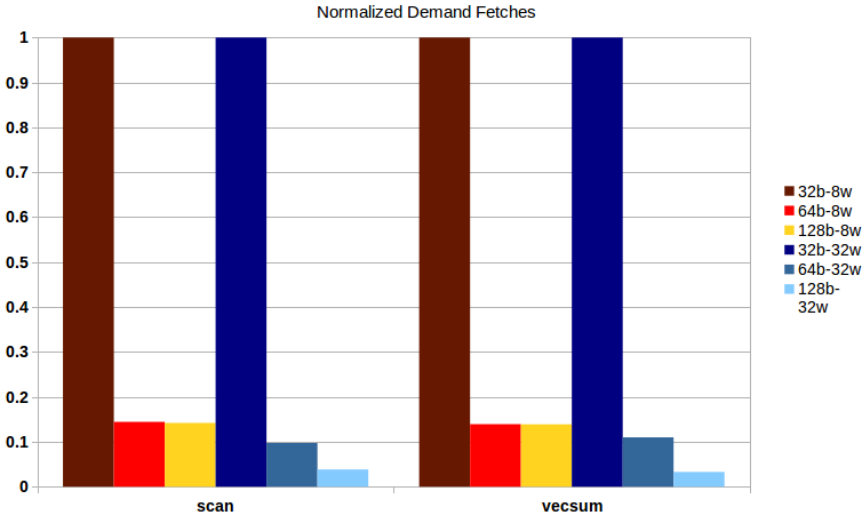

Figure 2: Normalized Demand Fetches

4 Mapping Scheme

For this section cache line size is 32 Bytes. The schemes are different in which bits they choose for coalescing. The main metric I use to select coalescing on which bits are more beneficial for an applications is the number of coalesced request versus number of non coalesced request. After a study on different bits for coalescing, for both applications I found mapping based on **bits 19 12** gives the most coalesced accesses. In figure 3 I have shown the cache performance metrics with the base line of coalescing with lower bits and no coalescing. Basic Coalescing miss rate is higher since it just reduces number of demand fetches but the ratio stays the same.

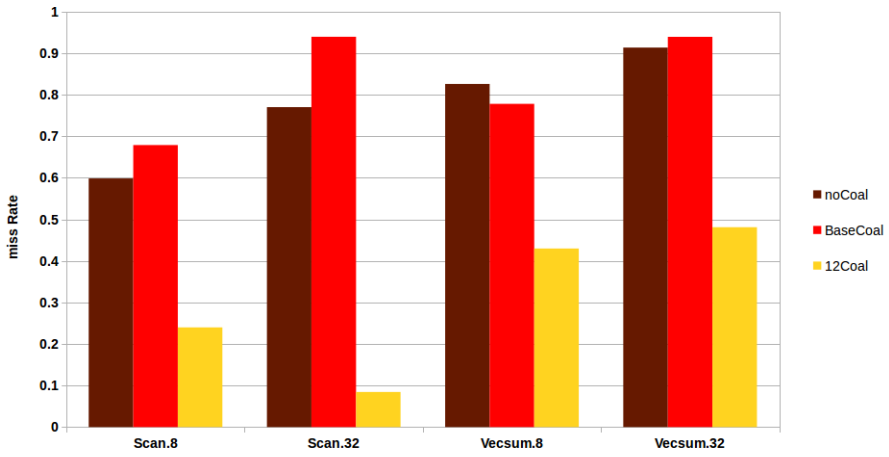

Figure 3: Mapping Schemes Miss Rates