Low Level Optimization by Data Alignment

Presented by:
Mark Hauschild

Motivation

- We have discussed how to gain performance
  - Application already done, send it off to grid
- Switch gears this class
- Low-level optimization
  - What can we do to our code to speed it up
  - Data alignment issues
- “It is impossible to efficiently process large-scale arrays without taking into account specific features of the DRAM architecture”

Outline

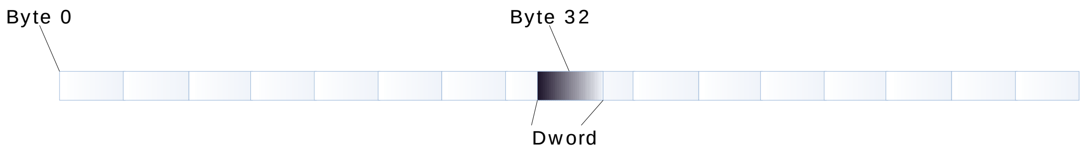
- Data Alignment Basics
- Manual Data Alignment
- Aligning Data Flows
- Aligning Byte-Data Flows
- Within a cache line
- Summary

Data Alignment Basics

- Processing arrays is a very common task
- We usually access data in small chunks
  - Value of A[8], possibly 4 bytes
- Smallest it reads is line size of L2 cache
  - 32, 64, 128 bytes
- Does not allow arbitrary addresses
  - Must start at a multiple

Data Alignment Basics

- So what happens if we try to access a value at address 30?



- Now must read two lines in the cache

Data Alignment Basics

- So what are the effects?
  - If reading sequentially, not a huge loss
    - Have to read the data anyway
    - but still extra cycle to combine
  - If not, doubling our memory overhead
  - Very large overhead when writing
    - But only to cache

Data Alignment Basics

- Most tools wont work
  - Even if they do, only do it by 16 bytes
- Could resort to assembly (bad)
- Could read just bytes, but inefficient
- Instead, note C pointers are integers
  - Can work with them directly

Manual Data Alignment

- Allocate structures ourselves
  - Offset a pointer to align the data
- Get our offset using the formula

$$Y = (X / N) * N$$

- Y is closest multiple of N below X
  - If 30, then 0, if 33, then 32
- Can get rid of division using logical AND

Manual Data Alignment

- Some code

```
char p;
p = (char*) malloc(size + align – 1);
p = (char*)(((int)p + align – 1) & ~(align – 1));
```

- Now accesses to p will always be aligned
- Slight increase in memory

Manual Data Alignment

- Similar trick for static memory

```
#define size 1024
#define align 64
int a[size + align – 1];
int *p;
p=(int*)(((int)&a+align-1)&~(align-1));
```

- Pointer p is now at starting position of aligned portion

Aligning Data Flows

- What if we do not allocate it ourselves

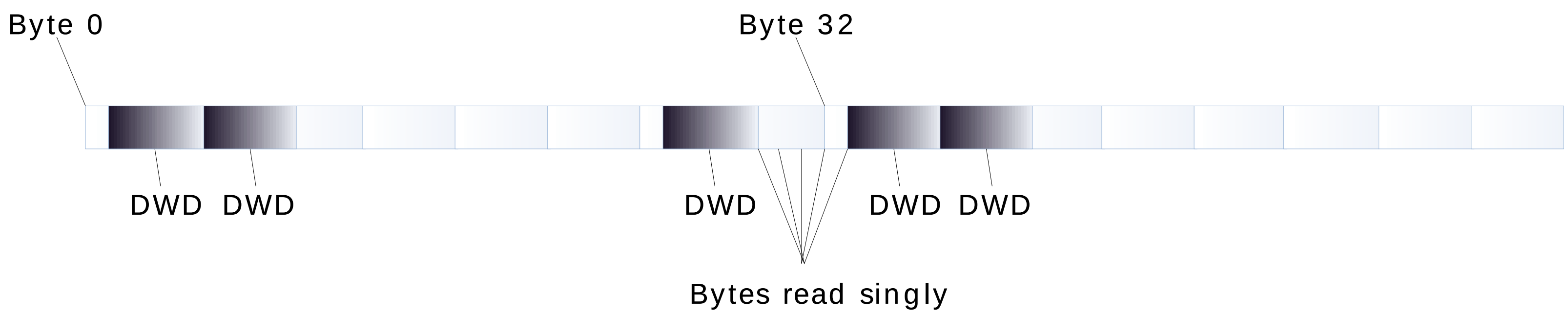
```
int sum(int *array, int n) {
   int a,x = 0;
   for (a=0; a < n; a++)
         x+= array[a];
   return x;
}
```

- No idea if it is aligned or not
- What do we do?

Aligning Data Flows

- Can still deal with it (with difficulty)
- Simple in theory
  - Read memory in our units until next read would cross boundary
    - Then read in bytes around boundary
    - Manually assemble it ourselves with shifts
    - Keep doing

Aligning Data Flows



- Problem is, if we use loops, inefficient
- Could use abunch of special cases
  - All unrolled
  - Pretty clunky
  - Can end up performing worse

Aligning Data Flows

- Example special case (one byte to right)

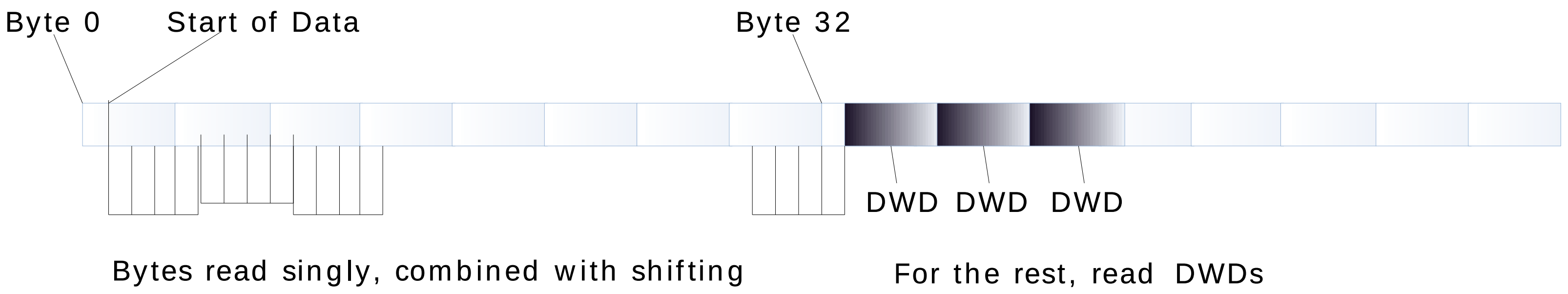
```
int sum_align(int *array,int n) {
int a,x=0;
char supra_bytes[4];
for(a=0;a<n;a+=8) {
    x += array[a+0];
    x += array[a+6];
    supra_bytes[0]=*((char*)array+(a+7)*sizeof(int)+0);
    supra_bytes[3]=*((char*)array+(a+7)*sizeof(int)+3);
    x += *(int *)supra_bytes;
}
```

Aligning Byte-Data Flows

- What if processing a byte-stream
- More efficient to read by Dwords
  - but might be unaligned stream
- Just break it up into two tasks
  - First read by bytes up to our boundary
  - Then read by Dwords after
- Does not require special cases

Aligning Byte-Data Flows

- In this way we just benefit, lose nothing
  - Gain from using Dword
  - Avoid misalignment penalty



Within a cache line

- Single variables aligned in order declared
  - Following leaves 3 bytes floating

```
static int a;
static char b;
static int c;
static char d;
```

  - More efficient to do

```
static int a;
static int c;
static char b;
static char d;
```

Within a cache line

- It is deeper than this though
  - Cache banks are 32, 64, 128 bits
  - Better if two variables in separate banks
    - Assignment is one clock cycle
  - Maybe best to place all data in addresses of multiples of four
    - More synchronous operations possible
    - Problem: Might take up so much more memory, now out of cache space! Net loss

Summary

- Alignment matters for optimal efficiency
  - Especially with arrays, loop counters
- Some things can be done fairly easily
- However, some fixes are hard and could backfire
- If in doubt, profile and find hotspots

Any questions?