# Memory Models for HPC

David Chisnall

February 14, 2011

# The Memory Hierarchy: Motivation

- Fast memory is expensive
- Slow memory is cheap

# Example Memory Hierarchy

| Location | Size | Access Time (cycles) |
|---|---|---|
| Registers | $<$ 1KB | 1 |
| L1 Data Cache | 32KB | 4 |
| L2 Cache | 256KB | 12 |
| L3 Cache | 2MB+ | 26-32 |
| Main Memory | 1GB+ | 150+ |

# Cache and Working Sets

- Most programs have a small *working set*
- Each part of the program works on a small amount of data
- Storing the working set in fast memory gives the same speed as storing everything in fast memory
- (But is much cheaper)

# What goes in Cache?

- Memory that was accessed recently
- Memory near memory that was accessed recently
- Memory that will be accessed soon (hopefully!)

# Cache Design

- Organised in *cache lines*, typically 64 bytes (8-16 pointer-sized values)
- Simple hash table indexed by some bits in the address

# Why do we care?

- Cache is from the French meaning ‘hidden’ — programmers don’t normally see it
- We do, because we care about performance!
- Line size means that it’s usually very fast to access something within 32 bytes of the last thing we accessed
- Hash conflicts mean that we can evict stuff from cache too soon

# Explicit Prefetching

- Working out what data will be used next is hard!
- The CPU tries to guess based on linear access (sometimes)
- The compiler can usually make better guesses
- The programmer can make even better ones

# Prefetching Example

```
for (int i=0 ; i<pixelCount ; i++)
{
    __builtin_prefetch(&pixels[i+1]);
    pixels[i].r *= 1.1;
    pixels[i].g *= 1.1;
    pixels[i].b *= 1.1;
}
```

- `__builtin_prefetch()` is a GCC extension
- Expands to prefetch instruction
- This prefetches the pixel needed for the next loop

# Is This Sensible?

```
__builtin_prefetch(&pixels[i+1]);
```

- Pixel is 1/4 of a cache line (16 bytes)
- Next pixel is in the current cache line 3/4 of the time
- Prefetch will take 12+ cycles
- We aren't giving it that long
- Prefetching pixel `i+4` is better

# Cache Optimizing: Hopscotch Hash Algorithm

1. Try to insert value at hash
2. If it's full, linear scan up to 32 elements along for spare slot, insert if possible
3. If not, see if any of the 32 values can be moved to a secondary location, insert in their free'd slot
4. If not, resize the table and retry

# Hopscotch Hash Performance

- Values always within 32 spots of first lookup
- Usually in same cache line
- Always within 2 cache lines
- Very fast!
- Also well-suited to concurrency

# The Multicore Problem

- Each core has its own cache
- Two threads try to access the same data
- What happens?

# Cache Coherency

- Two cores reading the same data, both have copy in cache
- One tries to modify, must invalidate other cache first
- Getting data from another core's cache is typically 40-60 cycles
- For best performance, threads should operate on different data

# Going Further: NUMA

- Other parts of the memory hierarchy see the same problem
- CPU has local memory, accesses other processors' memory indirectly
- Common example: Modern x86 chips
- Extreme example: SGI Altix

# Altix Pointers

- 36-bit node offset (64GB per node)
- 11-bit node ID
- Main memory used to cache remote node's memory
- Cache coherency problem across nodes, not just cores
- Similar problem on all *single system image* clusters

# Look, No Cache: Cell

- Cell SPU cores have 256KB of local memory
- Same idea as cache, but explicitly managed
- Programmer copies between it and main memory
- Tries to do work entirely in local memory
- Very fast, not so easy to use

# No Cache: Stream Processors

- Caches work well with certain categories of problem
- Very badly with others
- Some algorithms run on entire data set...
- ...but in a predictable sequence
- Stream processors are optimised to deliver the entire contents of memory to the program quickly in order
- Typically also have some local memory
- Examples: Most DSPs, GPUs

# Questions / Feedback?

- Is the speed okay? Too fast? Too slow?
- Is the material too abstract? Too specific?