Lecture 2: Single-processor Computing

CMSE 822: Parallel Computing

Prof. Sean M. Couch

Exercise 1.1

Exercise 1.1. Let us compare the speed of a classical FPU, and a pipelined one. Show that the result rate is now dependent on n: give a formula for $r(n)$, and for $r_\infty = \lim_{n\to\infty} r(n)$. What is the asymptotic improvement in r over the non-pipelined case?
Next you can wonder how long it takes to get close to the asymptotic behaviour. Show that for $n = n_{1/2}$ you get $r(n) = r_\infty/2$. This is often used as the definition of $n_{1/2}$.

On FPU: n results takes: $t(n) = n\ell\tau$ { ℓ : number of stages; τ : clock cycle time

rate of results: $r_{serial} \equiv \left(\frac{t(n)}{n}\right)^{-1} = (\ell\tau)^{-1}$

Pipelined: $t(n) = [s + \ell + n - 1]\tau$, s: setup cost

$= [n + n_{1/2}]\tau \rightarrow n_{1/2} = s + \ell - 1$

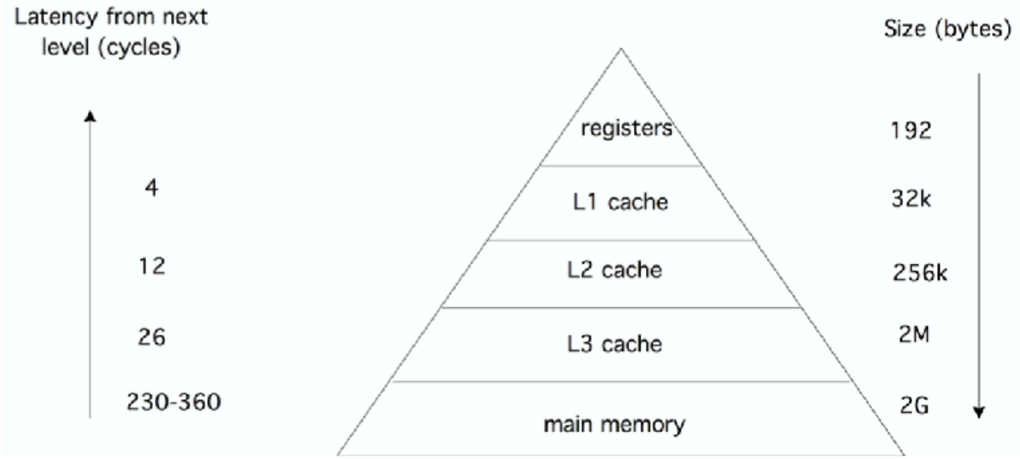
$r_{pipe} = \left(\frac{[n+n_{1/2}]\tau}{n}\right)^{-1} = \left(\tau + \frac{n_{1/2}\tau}{n}\right)^{-1} \rightarrow r_\infty = \tau^{-1}$

new let $r(n) = r_\infty/2 \Rightarrow \frac{t(n)}{n} = 2\tau \Rightarrow n_{1/2} = \frac{t(n)}{2\tau}$

Exercise 1.5

Exercise 1.5. The L1 cache is smaller than the L2 cache, and if there is an L3, the L2 is smaller than the L3. Give a practical and a theoretical reason why this is so.

The L1 cache has lower latency and higher bandwidth and is therefore more expensive to manufacture. Also, suppose the L2 was smaller than the L1, then there would never be reuse of data out of it: it is not possible that data would be in L2 but not be in L1.



Why not have more levels of cache? Diminishing returns…

Exercise 1.12

Exercise 1.12. Consider two processors, a data item x in memory, and cachelines x_1,x_2 in the private caches of the two processors to which x is mapped. Describe the transitions between the states of x_1 and x_2 under reads and writes of x on the two processors. Also indicate which actions cause memory bandwidth to be used. (This list of transitions is a *Finite State Automaton (FSA)*; see section 18.)

- When a processor reads from memory location x, the data item x is first checked in the processor's cache. If x1 (in the first processor's cache) is in a valid state and contains the data for memory location x, the processor will read the data from x1 and the state of x1 remains unmodified. This action does not cause memory bandwidth to be used.

- If x1 is not in a valid state or does not contain the data for memory location x, the processor will read the data from memory location x into x1 and update x1 to a valid state. This action causes memory bandwidth to be used.

- When a processor writes to memory location x, the data item x is first checked in the processor's cache. If x1 (in the first processor's cache) is in a valid state and contains the data for memory location x, the processor will update the data in x1 and mark x1 as dirty. This action does not cause memory bandwidth to be used.

- If x1 is not in a valid state or does not contain the data for memory location x, the processor will write the data to memory location x and update x1 to a valid state and mark it as dirty. This action causes memory bandwidth to be used.

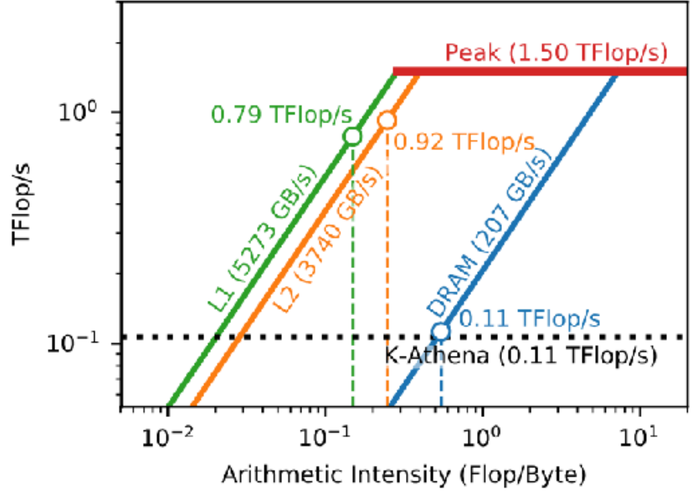
- Similarly, the transitions and actions will occur for x2 (in the second processor's cache) when reading and writing to memory location x on the second processor.

- It's important to note that if one processor writes to the memory location x and updates its cache x1 and the other processor also has a copy of that memory location in its cache x2, the x2 copy will be considered as invalid and the next time the processor try to access it, it will have to bring it from the main memory to the cache.

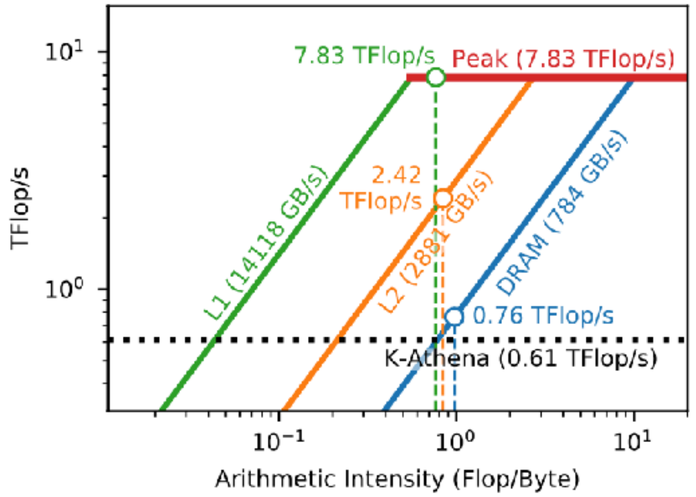
Exercise 1.15

Exercise 1.15. How would you determine whether a given program kernel is bandwidth or compute bound?

Profiling!



(a) CPU Roofline



(b) GPU Roofline

Fig. 2. Roofline models of a 2 socket Intel Xeon Gold 6148 "Skylake" CPU node on NASA's Electra (2a) and a single NVIDIA Tesla V100 "Volta" GPU on ORNL's Summit (2b). For both cases shown here and all other architectures we tested, DRAM bandwidth (or MCDRAM bandwidth for KNLs) is the limiting bandwidth for K-ATHENA's performance.

Grete et al. 2019, IEEE TPDS

Exercise 1.21

Exercise 1.21. Give an example of a doubly-nested loop where the loops can be exchanged; give an example where this can not be done. If at all possible, use practical examples from this book.

Independent: matrix-vector product

Dependent: successive over-relaxation iteration (i.e., Gauss-Seidel)

PCA Questions

Division vs. multiplication

- Division algorithms are *iterative*. Typically at least 2x more expensive.
- Should you avoid division? Yes, if you can… (division by true variable). Compile will try to help. Try. BUT! Never let optimization slow you down ;-)
- See Wikipedia entry on division algorithms

PCA Questions

Section 1.7.2 talks about loop unrolling, in order to increase the number of operations per second. Is this sort of optimization something a compiler can do? Also, in Table 1.2, there's a large reduction in the cycle time; for more complicated loops, is there a similar speedup?

- Can compiler unroll? Yes! See this StackOverflow response.
- More complicated loops? Should see similar speed up, maybe better since there will be more work in the inner loop. Watch out for loop dependencies, though!

PCA Questions

Is there a way for a program to tell how large the L1 cache is and make adjustments accordingly? If there were a magic function int l1Size = getL1Size(), you could use that pretty effectively I imagine.

- Not during runtime, really. Need to profile.
- During compilation, or pre-compilation, can set or detect size of cache for specific architecture and make automated optimizations accordingly
- See “cache oblivious” algorithms.

PCA Questions

In Page 57, it says "we note that we use the vectors sequentially, so, with a cacheline of eight doubles, we should ideally see a cache miss rate of 1/8 times the number of vectors m". How does this 1/8 calculated?

- double is 8 bytes, so 8 fit into a 64K cache line
- after every eight entries in the vector, a cache miss will occur

In-class Group Exercise

Spatial and temporal locality

Consider the following pseudocode of an algorithm for summing n numbers $x[i]$ where n is a power of 2:

```
for s=2,4,8,...,n/2,n:
  for i=0 to n-1 with steps s:
    x[i] = x[i] + x[i+s/2]
sum = x[0]
```

Analyze the spatial and temporal locality of this algorithm, and contrast it with the standard algorithm

```
sum = 0
for i=0,1,2,...,n-1
  sum = sum + x[i]
```

In-class Group Exercise

Spatial and temporal locality

Solution to exercise 1.14. *The standard algorithm has only spatial locality through the reuse of cache lines. The other algorithm only reuses cache lines when s is less than the cache line size. On the other hand it has cache reuse since elements are accessed multiple times: if $n/(s/2)$ is less than the cache size, elememts will fit in cache for all subsequent values of s.*

In-class Group Exercise
Reuse

Exercise 1.17. Consider the following code, and assume that `nvectors` is small compared to the cache size, and `length` large.

```
for (k=0; k<nvectors; k++)
  for (i=0; i<length; i++)
    a[k,i] = b[i] * c[k]
```

How do the following concepts relate to the performance of this code:

- Reuse
- Cache size
- Associativity

Would the following code where the loops are exchanged perform better or worse, and why?

```
for (i=0; i<length; i++)
  for (k=0; k<nvectors; k++)
    a[k,i] = b[i] * c[k]
```

In-class Group Exercise

Reuse

Solution to exercise 1.15. *Reuse.*

- *Each element of the array* `a` *is touched only once, so there is no reuse possible.*
- *Arrays* `b` *and* `c` *are used multiple times, so the algorithm has possibility for reuse.*

Cache size.

- *The array* `b` *is used multiple times, but between two uses of an element, each other element is touched, so if the array is larger than the cache, there will be no temporal locality and no reuse.*
- *(The array* `b` *is used sequentially, so each cache line that is loaded is fully used. In other words, there is spatial locality.)*
- *Each element of the array* `c` *is used multiple times in the inner loop, and the uses of a single element are just a few operations apart, so there is temporal locality, and the elements will be reused in cache.*

Associativity: whenever you traverse two arrays you have to worry about cache thrashing, meaning that the arrays get mapped to the same location in cache. That problem can exist here with `a[k,*]` *and* `b`.

The second code:

- *The same element of* `b` *is used in the inner loop, so there will be full reuse.*
- *The array* `c` *is much shorter, so it will probably remain in cache.*
- *There may be a problem with the cachelines of* `a`*, since of each cacheline only one element is used. However, since the number of cachelines of* `a` *that is loaded is low, they may still be fully used.*

Group work on Project 1!