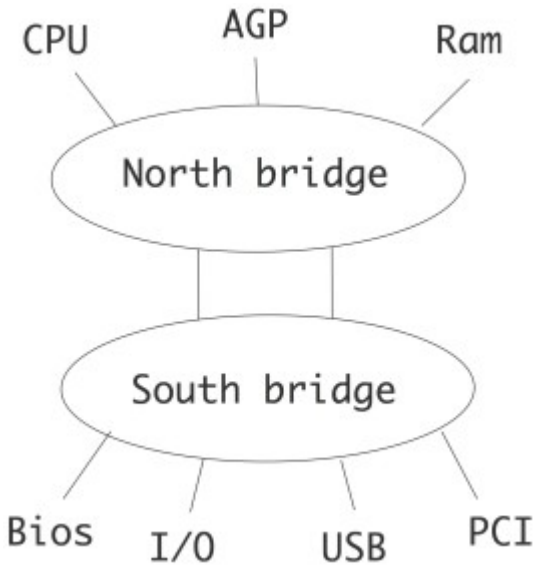
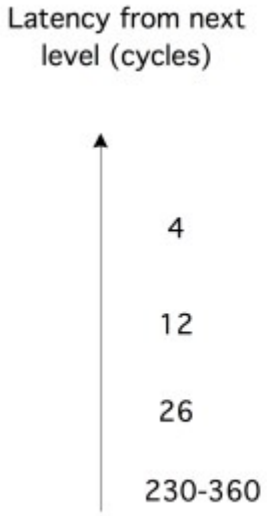
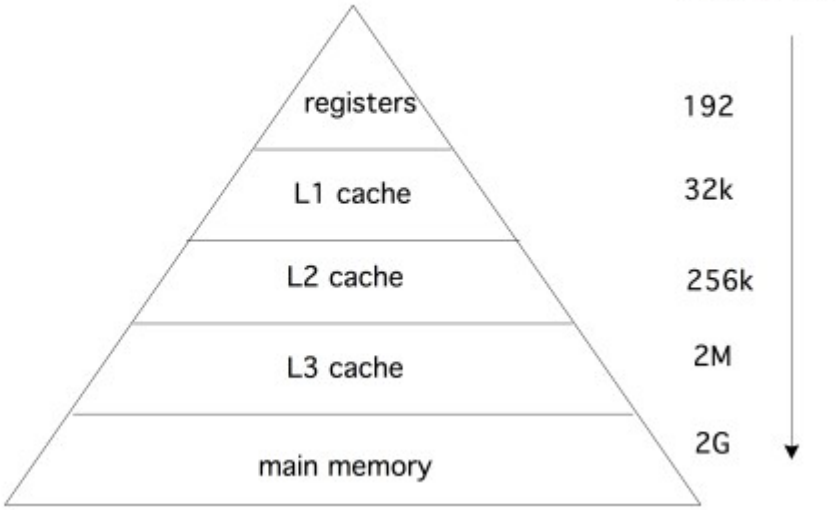
Recall Memory Hierarchy







Hardware Concerns

- For truly HPC, we need to consider the hardware and architecture on which our code is going to operate
  - How much of each type of memory is available?
  - Use this information to modify loops and data flow operations
  - Massive gainz!

Caches

- Built “in between” the registers and main memory (RAM)
- Higher bandwidth and lower latency than main memory, by a large margin
- Main idea: if some piece of memory is used multiple times, keep it in the cache
  - Downside: we need to check the cache and know what’s already in there
- Typically outside control of code, but we can still take it into account!

Caches

- Also have the overhead of cache tags, used to reconstruct the original memory location and information about data
- Comes in various sizes, typically L1, L2, etc., in decreasing order of performance and, inversely, size
- Try the `lscpu` command in linux to get some CPU info
- Be careful when benchmarking, L3 can be very large (10s of megabytes!)

Cache loads and misses

- Compulsary miss: the first time we use data, it is definitely going to miss
  - Hardware can still try and predict!
- Capacity miss: data was overwritten because we filled the cache with newer data
  - To avoid: partition access into “chunks”
- Conflict miss: two data get mapped to the same location, but there were better candidates for eviction
- Invalidation miss: data is invalid, another (parallel) computation changed it in RAM

Locality and Data Reuse

- We want to analyze when algorithms even have the ability to leverage data reuse
- Example: vector addition has no data reuse because each element is only used once :(
- Example: vector-matrix multiplication definitely has some re-use because the vector is used once for each row of the matrix!
  - Still need to consider the size of each to make sure we don’t have capacity miss

Cache replacement policies

- Least Recently Used (LRU) tracks the last time each was used, replacing the oldest first
- FIFO removes the least recently saved data first
- Random replacement flushes randomly
  - Not as bad as you think!
- Removal is called a “flush”

Types of Caches

- Need for mapping: where to put data from cache back into RAM?
- Direct map: e.g. for 32-bit mem address, and cache can hold 8K words (i.e. 64kB), meaning 16 bits to address. So direct mapping will take last 16 bits from physical memory address and remember in the cache
  - Problem, if items are a whole cacheline apart, we get a conflict miss!
  - Benefit: very quick to compute

Associative caching

- For some k > 1, map elements in such a way that an element can appear in any of k different cache locations
- This means, we need a k+1-way collision for a conflict miss
- So for k large enough, relative to the data, we may never get a conflict!
- Bad news: need to check k places to find data in the cache, and need to do more computation beforehand

Cache Blocking

- Idea: stride across arrays in ways that allow more L1 cache usage

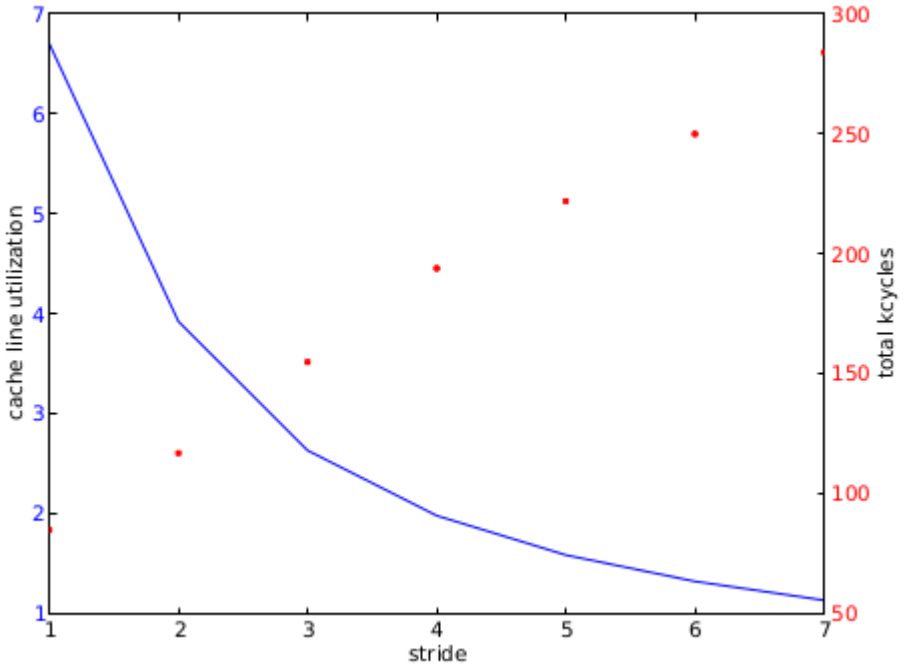
```
for (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[j]+1.2;
```

```
for (b=0; b<size/l1size; b++) {
  blockstart = 0;
  for (i=0; i<NRUNS; i++) {
    for (j=0; j<l1size; j++)
      array[blockstart+j] = 2.3*array[blockstart+j]+1.2;
  }
  blockstart += l1size;
}
```

Using Cachelines

- Better to stride less than the cache line

```
for (i=0,n=0; i<L1WORDS; i++,n+=stride)
  array[n] = 2.3*array[n]+1.2;
```



Loop Tiling

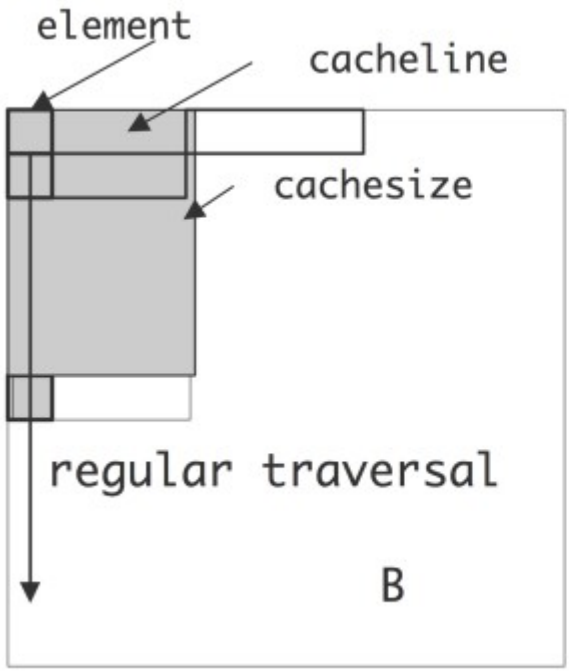
- Consider matrix A = A + B^T:

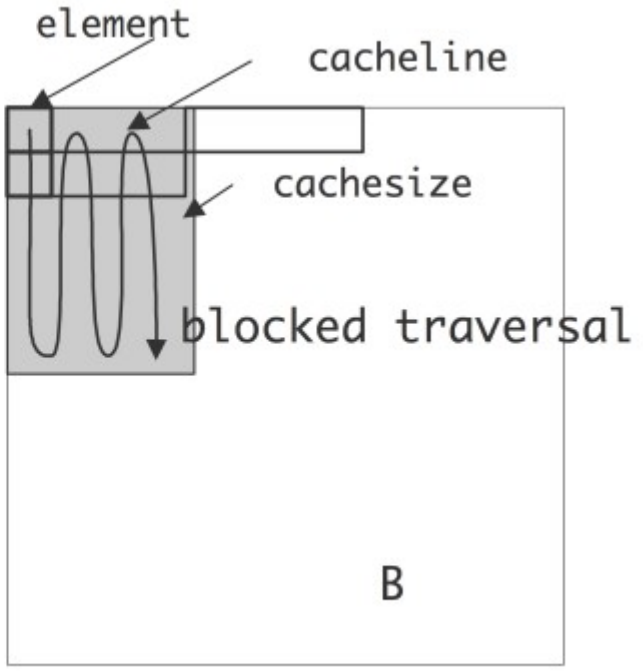
```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    A[i][j] += B[j][i];
```

- Versus the “tiled” version:

```
for (int ii=0; ii<N; ii+=blocksize)
  for (int jj=0; jj<N; jj+=blocksize)
    for (int i=ii*blocksize; i<MIN(N,(ii+1)*blocksize); i++)
      for (int j=jj*blocksize; j<MIN(N,(jj+1)*blocksize); j++)
        A[i][j] += B[j][i];
```

Loop Tiling


element
cacheline
cachesize
regular traversal
B


element
cacheline
cachesize
blocked traversal
B

Tiling Matrix Multiplicaion

- for kk=1..n/bs

```
    for i=1..n
        for j=1..n
            S = 0
            for k=(kk-1)*bs+1..kk*bs
                s += a[i,k]*b[k,j]
            c[i,j] += s
```

- Keeps rows of A in the cache for faster code

Fastest MM comparison