CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Cache

DRAM is much slower than CPU

- hundreds of CPU cycles to fetch a **64-byte cache line**

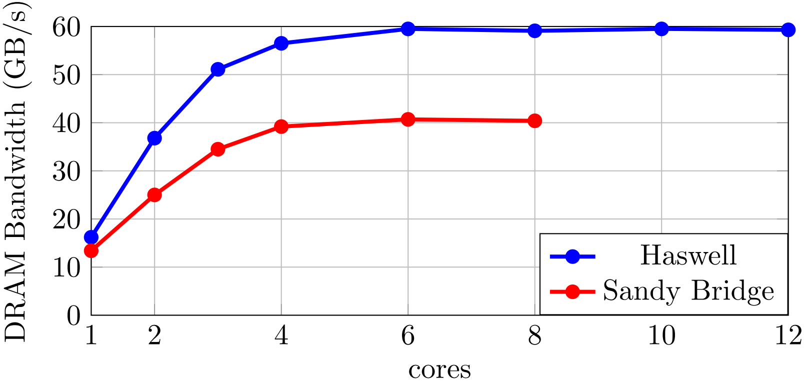
CPUs hide this latency using **cache**

- cache is high-speed memory on the CPU die
- there are multiple levels of cache on the Sandy Bridge
  - L1 : 32 KB data, 32 KB instruction **per core**
  - L2 : 256 KB **per core**
  - L3 : 20 MB **shared by all cores**
- recently-used data is retained in cache
  - a line in cache is evicted when a cache line is fetched
  - the hows and whys of this are complicated!
- the key optimization is to maximize cache use
  - use all information in a cache line
  - avoid loading cache lines more than once

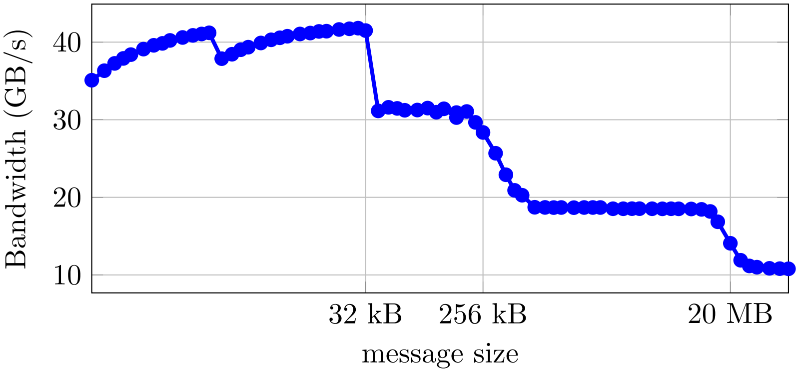
DRAM Bandwidth

triad stream benchmark

```
#pragma omp parallel for
#pragma omp simd
for (j=0; j<n; j++)
  a[j] = b[j]+scalar*c[j];
```



Cache Bandwidth: Single Core



Cache-Aware Example: Double blur

One-dimensional blur kernel

$$\text{out}_i \leftarrow 0.25(\text{in}_{i-1} + 2\text{in}_i + \text{in}_{i+1})$$

- each output value is a linear combination of neighbours in input array

Implementation of blur kernel

```
void blur(double *in, double *out, int n){
  for(auto i=1; i<n-1; ++i)
    out[i] = 0.25*(2*in[i]+in[i-1]+in[i+1]);
}
```

Buffering

A pipelined workflow uses the output of one “kernel” as the input of another

- on the CPU these can be optimized by keeping the intermediate result in cache for the second kernel

An example is applying the double blur kernel twice

Double blur

```
void blur_twice(const double* in , double* out , int n) {
  static double* buffer = malloc(n, sizeof(double)*n);

  for(auto i=1; i<n-1; ++i) {
    buffer[i] = 0.25*(in[i-1] + 2.0*in[i] + in[i+1]);
  }
  for(auto i=2; i<n-2; ++i) {
    out[i] = 0.25*(buffer[i-1] + 2.0*buffer[i] + buffer[i+1]);
  }
}
```

Naiive Implementation

Let's use OpenMP

- parallelize each for loop with an OpenMP directive

Double blur: basic parallelization

```
void blur_twice(const double* in , double* out , int n) {
  static double* buffer = malloc(n, sizeof(double)*n);

  #pragma omp parallel
  {
    #pragma omp for
    for(auto i=1; i<n-1; ++i) {
      buffer[i] = 0.25*(in[i-1] + 2.0*in[i] + in[i+1]);
    }
    #pragma omp for
    for(auto i=2; i<n-2; ++i) {
      out[i] = 0.25*(buffer[i-1] + 2.0*buffer[i] + buffer[i+1]);
    }
  }
}
```

What happens when `n` is so large that `buffer` can't fit into cache?

Cache-Aware Implementation

- each thread has a private buffer
- the domain is broken into blocks that fit into cache
- 2.8 × speedup

Double blur: cache friendly

```
void blur_twice(const double* in , double* out , int n) {
  auto const block_size = std::min(512, n-4);
  auto const num_blocks = (n-4)/block_size;
  static double* buffer = malloc_host<double>((block_size+4)*
      omp_get_max_threads());

  #pragma omp parallel for
  for(auto b=0; b<num_blocks; ++b) {
    auto tid = omp_get_thread_num();
    auto first = 2 + b*block_size;
    auto last = first + block_size;

    auto buff = buffer + tid*(block_size+4);
    for(auto i=first-1, j=1; i<(last+1); ++i, ++j) {
      buff[j] = 0.25*(in[i-1] + 2.0*in[i] + in[i+1]);
    }
    for(auto i=first, j=2;   i<last;   ++i, ++j) {
      out[i] = 0.25*(buff[j-1] + 2.0*buff[j] + buff[j+1]);
    }
  }
}
```

Exercises: Cache Awareness

The example code implements a dispersion kernel:

- applying the Laplacian twice to each point in the domain
- one of the key kernels in the dynamical core of atmospheric models

dispersion kernel

```
for i = 1:nx-1
  for j = 1:ny-1
    lap(i,j) = -4*in(i,j) + in(i-1,j)   + in(i+1,j)
                          + in(i,j-1) + in(i,j+1);
  end
end
for i = 2:nx-2
  for j = 2:ny-2
    out(i,j)
      = in(i,j) - D*(-4*lap(i,j) + lap(i-1,j) + lap(i,j+1)
                                 + lap(i,j+1) + lap(i,j-1))
  end
end
```

Exercises: Cache Awareness

1. open `topics/openmp/practicals/cxx/dispersion.c`
2. what is the difference between each kernel implementation?
3. which do you think will be the fastest?
4. compile with a vectorization report
5. run the test script and look at the output

Double blur: basic parallelization

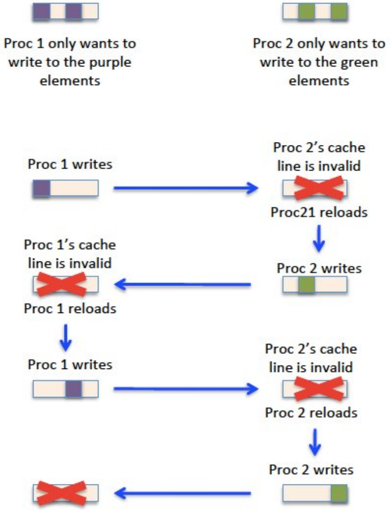
```
cc -h list=a dispersion.c -O3
vim laplace.lst
./test_dispersion.sh
```

Cache Coherency

- Caches are copies of global memory
  - multiple cores could hold the same cache lines in their cache
  - not likely for different MPI ranks
  - very likely for OpenMP applications
- Before accessing memory, the processor has to ensure that the values in all copies of a cache line are **consistent**
  - called **cache coherency**
  - multiple cores could hold the same cache lines in their cache
  - if one core has modified its copy of a cache line, the other cores need to update their copy before making their own modifications
  - this process has a high performance overhead
  - GPUs are not cache coherent
- Values stored in CPU registers do not have to be consistent
  - hence race conditions

Contention: Thrashing & False Sharing

- Cache coherency is maintained at cache line granularity
- Even if threads update different values, if the values are in the same cache line, the cache line will be swapped between cores
- **Worst case**: false sharing causes slow down relative to serial code



Avoiding False Sharing

- False sharing occurs when both
  1. multiple threads **write** to shared data that is in the same cache line
  2. data accesses to the same cache line from multiple threads have close **temporal proximity** and occur **multiple times**
- Use thread-private copies of data
- Data that is more than one cache-line away from data that any other thread will access will not lead to false sharing
- data that is only read and not written cannot lead to false sharing
  - another good reason to write const-safe code in C++

Exercise: False Sharing

- can you find the false sharing **performance bug** in `topics/openmp/practicals/cxx/histogram.cpp`