OpenMP Optimization Vectorization & Caches

Ben Cumming, CSCS

July 19, 2016

Overview

The aim of this section is to give a brief introduction to the two most important optimization methods on multicore CPUs

- vectorization
- cache utilization

Trends

To improve computational performance of a processor there are three options:

1. ~~increase clock frequency~~
2. increase work per clock cycle
   - more cores
   - **vectorization**
3. don't stall or wait
   - use **caches** to reduce memory latency
   - branch prediction

Power increases super-linearly with frequency

- clock frequencies will not increase in the forseeable future

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Vectorization

Degrees of Parallelism

There are three main "forms" of parallelism available on multicore HPC systems:

- **network** parallelism between nodes/sockets (e.g. MPI)
- **multicore** parallelism between cores in a socket (e.g. OpenMP)
- **vector** units in each core (e.g. Intel AVX)

All three must be utilized!

Vectorization

Vectorization performs multiple operations **in parallel** on a core with a single instruction

- data is loaded into **vector registers** that are described by their width in bits
  - 256 bit registers : 8× `float` **or** 4× `double`
  - 512 bit registers : 16× `float` **or** 8× `double`
- **vector units** perform arithmetic operations on vector registers simultaneously
- some architectures can perform multiple vector operations simultaneously
  - e.g. Intel Haswell can perform 2 FMA (multiply + add) simultaneously
    - i.e. 2 ops × 2 (add + mul) × 4 double = 16 DP ops/cycle

Vectorization is key to maximising computational performace

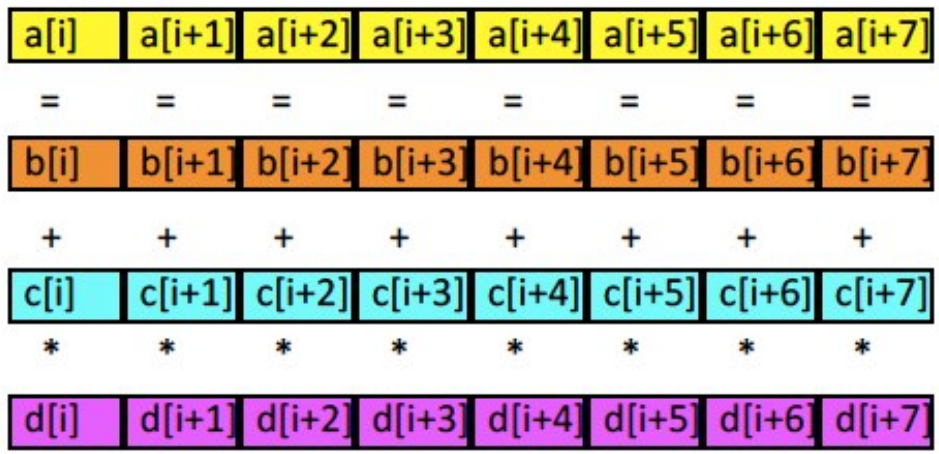
Vectorization Illustrated

sequential

```
a [i] = b[i] + c[i] * d[i];
```

8×vectorization

```
a [i:8] = b[i:8] + c[i:8] * d[i:8];
```



How to Vectorize

- use **vector intrinsics**
  - explicit hardware-specific instructions
  - high peformance
  - non-portable and hard to maintain
- automatic compiler vectorization
  - compiler will vectorize where it is possible
  - compilers can do a poor job
- use libraries that are already vectorized
  - let somebody else do the work for you

Vectorization Restrictions

The compiler can only vectorize under certain conditions

- compilers are conservative:
  - will only vectorize if guarenteed that vectorization will not change the result
  - this is harder to prove than you would imagine
- loads and stores are on contiguous memory locations
  - not strictly true: some processors have gather-scatter load and store into vector registers
  - aligned and contiguous loads and stores are always better
  - compilers are not good at scatter-gather vectorization

this...

```
void vec_add(double *x,
             double *y, int n) {
    for(auto i=0; i<n; ++i) {
        x[i] += y[i];
    }
}
double *a = new double[1000];
vec_add(a+1, a, 1000-1);
```

is equivalent to

```
void vec_add(double *x, int n){
    for(auto i=0; i<n; ++i) {
        x[i+1] += x[i];
    }
}
double *a = new double[1000];
vec_add(a, 1000-1);
```

The compiler can't ensure that the vectors `x` and `y` do not address the same memory (i.e. that they alias)

- if there is aliasing, vectorized and unvectorized code may produce different results
- so it won't vectorize!

solution: promise no aliasing

```
#pragma ivdep
for(auto i=0; i<n; ++i) {
    x[i] += y[i];
}
```

solution: force vectorization

```
#pragma omp simd
for(auto i=0; i<n; ++i) {
    x[i] += y[i];
}
```

The compiler can't be certain that the stores don't alias. Can be fixed with the Intel compiler with `#pragma ivdep`:

indirect indexing

```
double *x, *y, *z;
int *p;
for(auto i=0; i<n; ++i) {
    x[p[i]] = y[i] + z[i];
}
```

loads and stores are not contiguous:

non unit stride

```
double *x, *y, *z;
for(auto i=0; i<n; i+=2) {
    x[i] = y[i] + z[i];
}
```

Does my Code Vectorize?

Good question!

- compilers can generate reports that summarise which loops vectorized
- you can ask for different levels of detail
  - e.g. only loops that failed to vectorize
  - e.g. whether to explain why a loop didn't vectorize
- the flags vary from compiler to compiler:
  - **Intel** : `-vec-report=n`, or `-opt-report=n`
  - **GCC**: `-ftree-vectorizer-verbose=n`
  - **Cray**: `-h list=a`
- Cray's reports are very nice!

You can also use the `disassemble` command in gdb, if you like reading assembly.

Exercises: Vectorization

1. go to `topics/openmp/practicals/cxx`
2. make a `git pull` to update the exercise
3. have a look at `laplace.c` (look for the TODO comments)
4. compile with a vectorization report
5. add OpenMP directives to both loops
6. make one of the loops vectorize
7. run on a range of mesh sizes and thread numberings

```
cc -h list=a laplace.c -O3
vim laplace.lst
export OMP_NUM_THREADS=1
srun -c8 -n1 --hint=nomultithread ./a.out 1000 1000 100
srun -c8 -n1 --hint=nomultithread ./a.out 1000 2000 100
export OMP_NUM_THREADS=8
srun -c8 -n1 --hint=nomultithread ./a.out 1000 1000 100
srun -c8 -n1 --hint=nomultithread ./a.out 1000 2000 100
```

Cache

DRAM is much slower than CPU

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

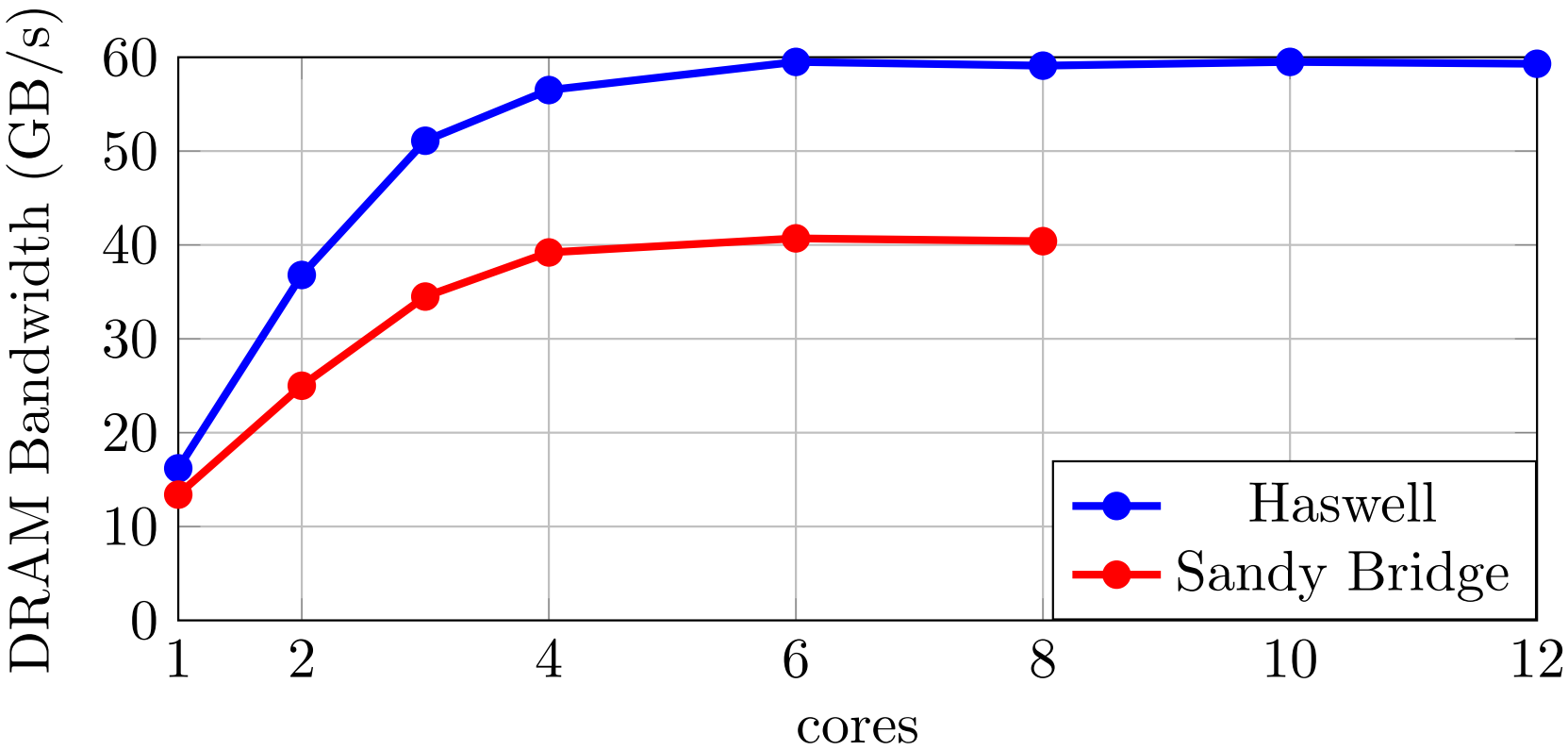
CPUs hide this latency using **cache**

- 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

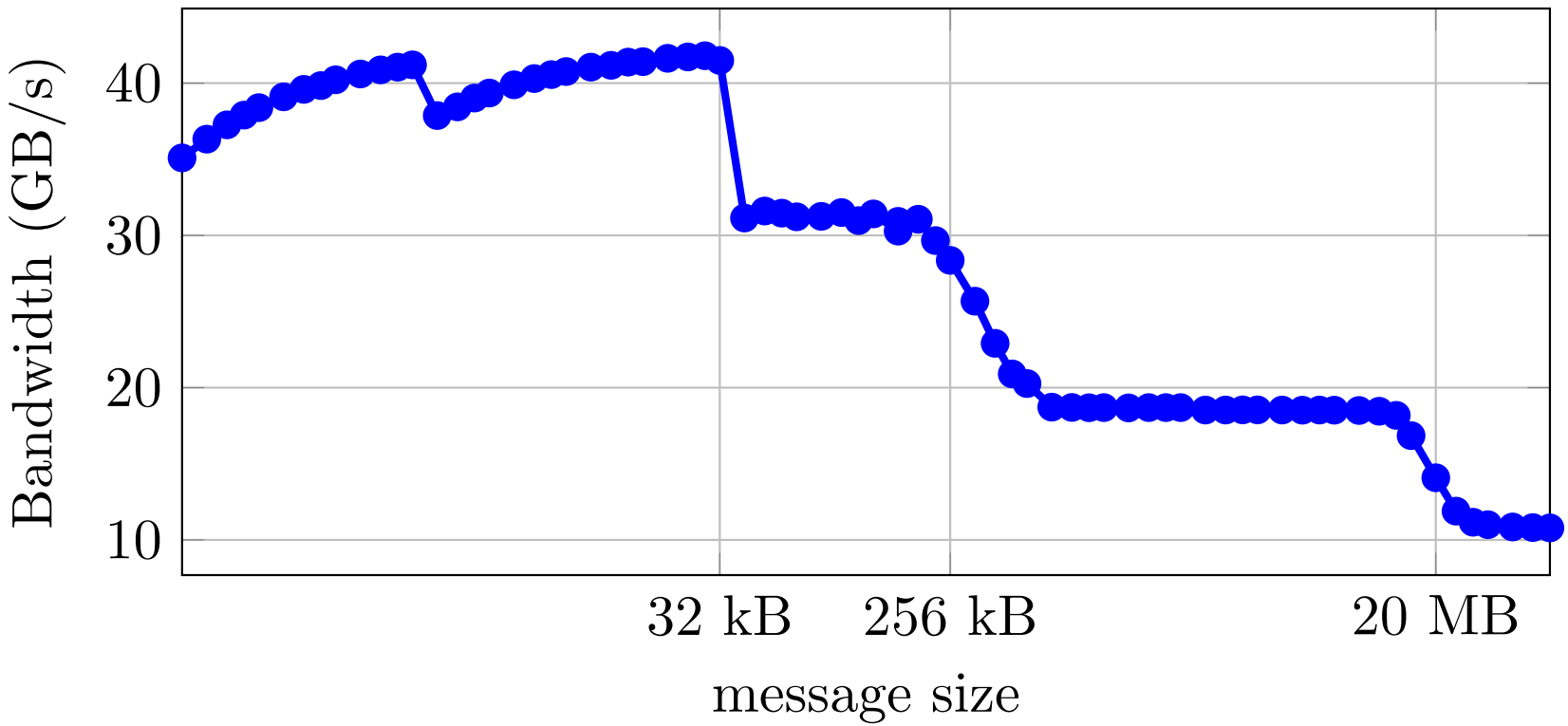
DRAM Bandwidth

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

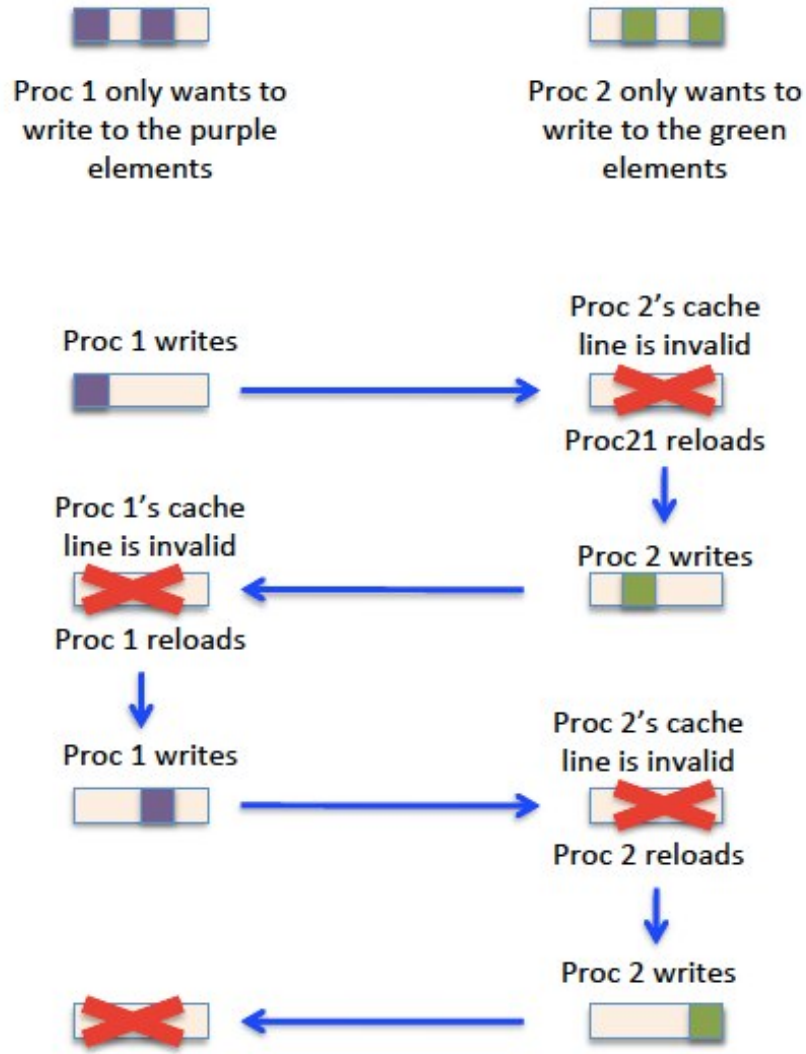
```
cc -h list=a dispersion.c -O3
vim laplace.lst
./test_dispersion.sh
```

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`