CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

OpenMP Advanced Topics

Summer School 2017 – Effective High Performance Computing

Vasileios Karakasis, CSCS

July 26, 2017

Overview

The aim of this presentation 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. Dont 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

Vectorization

Degrees of parallelism

Multiple levels of parallelism are available on multicore HPC systems

- Parallelism across nodes (e.g., MPI)
- Parallelism across sockets (e.g., MPI, NUMA-aware data placement, thread pinning)
- Thread-level parallelism across cores/threads of the same socket (e.g., OpenMP)
- Data parallelism inside the core (**SIMD** or **Vectorization**)
- Instruction-level parallelism (out-of-order execution)

Efficient codes must utilize all of them!

Vectorization

Vectorization performs multiple operations **in parallel** with a single processor instruction

- Data is loaded into **vector registers** that are described by their width in bits
  - 256 bit registers: 8$\times$ `float` or 4$\times$ `double`
  - 512 bit registers: 16$\times$ `float` or 8$\times$ `double`
- **Vector units** perform arithmetic operations on the elements of vector registers simultaneously
  - Modern processors usually have more than one floating point arithmetic units
  - Some processors can also perform FMA (Fused Multiply-Add) instructions
    - A processor with 512-bit vector lanes and 2$\times$ FMA units can perform up to 32 flops/cycle!

Vectorization is key to maximising computational performance

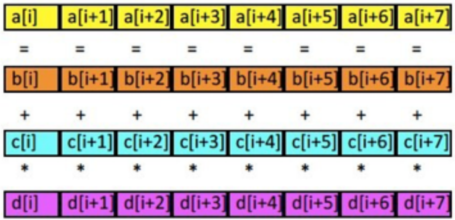
Vectorization Illustrated

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

1. Automatic compiler vectorization
   - Compiler will vectorize where it is possible
   - Compilers must be conservative, so they might not be so successful

How to Vectorize

1. Automatic compiler vectorization
   - Compiler will vectorize where it is possible
   - Compilers must be conservative, so they might not be so successful
2. Guided compiler vectorization
   - Help the compiler vectorize by making it feel safe with its optimizations

How to Vectorize

1. Automatic compiler vectorization
   - Compiler will vectorize where it is possible
   - Compilers must be conservative, so they might not be so successful
2. Guided compiler vectorization
   - Help the compiler vectorize by making it feel safe with its optimizations
3. Use libraries that are already vectorized
   - Let somebody else do the work for you

How to Vectorize

1. Automatic compiler vectorization
   - Compiler will vectorize where it is possible
   - Compilers must be conservative, so they might not be so successful
2. Guided compiler vectorization
   - Help the compiler vectorize by making it feel safe with its optimizations
3. Use libraries that are already vectorized
   - Let somebody else do the work for you
4. Use compiler **vector intrinsics**
   - 1-1 correspondence with actual hardware instructions
   - High performance (if you know exactly what you are doing)
   - Non-portable and hard to maintain

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
  - Scatter-gather vectorization is generally not so efficient

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 cannot 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 wont vectorize!

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 cannot 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 wont 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 dont 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 didnt vectorize
- the flags vary from compiler to compiler:
  - Intel: `-qopt-report=n`
  - GCC: `-ftree-vectorizer-verbose=n`
  - Cray: `-h list=a`
- Crays reports are very nice!

If you’re brave enough, you can dig into the generated assembly:

- Use compiler options (e.g., `-s` in GCC), debuggers, shell utilities (e.g., `objdump`)

Exercise: 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 -c12 -n1 --hint=nomultithread ./a.out 1000 1000 100
export OMP_NUM_THREADS=12
srun -c12 -n1 --hint=nomultithread ./a.out 1000 1000 100
```