Hardware-aware Performance Tuning in C/C++

Jiří Filipovič

Sep. 2021

Focus of the Lecture

We will learn how to optimize C/C++ code to get more performance from contemporary processors

- maximizing benefit from cache architecture
- writing code taking advantage of compiler auto-vectorization
- using multiple cores efficiently

We will not cover all interesting topics...

- only basic optimizations from each category
- no language-specific optimizations (inlining, proper usage of virtual functions etc.)
- no hardcore, assembly-level optimizations

Demonstration Example

We will demonstrate optimization methods using concrete example

- I have tried to find as simple as possible computational problem, which still expose a lot of opportunity for various optimization techniques

The code is not very abstract or generic

- in productivity-optimized programming, we want to hide how are algorithms performed, how are data stored etc.
- however, when optimizing code, we have to focus on implementation details, thus, code looks more "old school"
- in practice, usually very small fraction of source code is performance-critical, so different programming style for optimized code is not a problem

Electrostatic Potential Map

Important problem from computational chemistry

- we have a molecule defined by position and charges of its atoms
- the goal is to compute charges at a 3D spatial grid around the molecule

In a given point of the grid, we have

$$V_i = \sum_j \frac{w_j}{4\pi\epsilon_0 r_{ij}}$$

Where w_j is charge of the j-th atom, r_{ij} is Euclidean distance between atom j and the grid point i and ϵ_0 is vacuum permittivity.

Electrostatic Potential Map

Initial implementation

- suppose we know nothing about HW, just know C++
- algorithm needs to process 3D grid such that it sums potential of all atoms for each grid point
- we will iterate over atoms in outer loop, as it allows to precompute positions of grid points and minimizes number of accesses into input/output array

Electrostatic Potential Map

```
void coulomb(const sAtom* atoms, const int nAtoms,
    const float gs, const int gSize, float *grid) {
  for (int a = 0; a < nAtoms; a++) {
    sAtom myAtom = atoms[a];
    for (int x = 0; x < gSize; x++) {
      float dx2 = powf((float)x * gs - myAtom.x, 2.0f);
      for (int y = 0; y < gSize; y++) {
        float dy2 = powf((float)y * gs - myAtom.y, 2.0f);
        for (int z = 0; z < gSize; z++) {
          float dz = (float)z * gs - myAtom.z;
          float e = myAtom.w / sqrtf(dx2 + dy2 + dz*dz);
          grid[z*gSize*gSize + y*gSize + x] += e;
        }
      }
    }
  }
}
```

Benchmarking

We will benchmark codes on pretty average desktop system

- 4 cores
- AVX2 (256-bit vectors), no FMA

Guess speedup of original code :-).

Cache Memories

Why we have cache memories in modern processors?

- main memory is too slow (both latency and bandwidth) comparing to compute cores
- we can build much faster, but also more expensive memory
- cache is fast memory, which temporary keeps parts of larger and slower memories

Cache Implementation

How is it working?

- multiple levels (usually L1 and L2 private for core, L3 shared)
- accessed by cache lines (64 bytes on Intel architectures)
- when data are accessed, they are stored in cache and kept until cache line is needed for another data
- limited associativity (each cache line may cache only defined parts of main memory)
- parallel access into memory – cache lines must be somehow synchronized (broadcast, invalidation)

Optimization for Cache

Optimization for spatial locality

- access consequent elements
- align data to a multiple of cache line size
- otherwise only part of transfered data is used

Optimization for temporal locality

- when data element needs to be accessed multiple times, perform accesses in a short time
- otherwise it may be removed from cache due to its limited capacity or associativity

Omit inefficient usage

- conflict misses
- false sharing

Electrostatic Potential Map

```
void coulomb(const sAtom* atoms, const int nAtoms,
    const float gs, const int gSize, float *grid) {
  for (int a = 0; a < nAtoms; a++) {
    sAtom myAtom = atoms[a];
    for (int x = 0; x < gSize; x++) {
      float dx2 = powf((float)x * gs - myAtom.x, 2.0f);
      for (int y = 0; y < gSize; y++) {
        float dy2 = powf((float)y * gs - myAtom.y, 2.0f);
        for (int z = 0; z < gSize; z++) {
          float dz = (float)z * gs - myAtom.z;
          float e = myAtom.w / sqrtf(dx2 + dy2 + dz*dz);
          grid[z*gSize*gSize + y*gSize + x] += e;
        }
      }
    }
  }
}
```

Evaluation

We have compiled the code above with vectorization switched off (as we are interested in effects of memory access only)

- 31.6 millions of atoms evaluated per second (MEvals/s) using $256 \times 256 \times 256$ grid and 4096 atoms
- 164.7 Mevals/s by changing grid size to $257 \times 257 \times 257$

Interpretation

- strong dependence on input size indicates problems with cache associativity
- even 164.7 Mevals/s is not very good result, considering 8 floating point operations are performed in innermost loop

Spatial Locality

We are interested in the innermost loop

- it defines memory access pattern (i.e., which elements are accessed consequently)
- the innermost loop runs over `z`, which creates large memory strides in accessing `grid`
- when grid size is power of two, columns hits the same associativity region

Optimization

- we need to rearrange loops: the innermost loop should iterate through `x`

Spatial Locality

```
void coulomb(const sAtom* atoms, const int nAtoms,
    const float gs, const int gSize, float *grid) {
  for (int a = 0; a < nAtoms; a++) {
    sAtom myAtom = atoms[a];
    for (int z = 0; z < gSize; z++) {
      float dz2 = powf((float)z * gs - myAtom.z, 2.0f);
      for (int y = 0; y < gSize; y++) {
        float dy2 = powf((float)y * gs - myAtom.y, 2.0f);
        for (int x = 0; x < gSize; x++) {
          float dx = (float)x * gs - myAtom.x;
          float e = myAtom.w / sqrtf(dx*dx + dy2 + dz2);
          grid[z*gSize*gSize + y*gSize + x] += e;
        }
      }
    }
  }
}
```

Evaluation

Performance measurement

- 371.8 Mevals/s using $256 \times 256 \times 256$ grid and 4096 atoms (from 31.6 and 164.7 MEvals/s)
- no sensitivity to changing grid size (no cache associativity problem)
- much better spatial locality

Analysis of cache pattern

- each atom is applied to the whole grid
- poor temporal locality (grid is too large structure)

Temporal Locality

Atoms array is much smaller than grid

- we can rearrange loops to iterate over atoms in the innermost loop: z-y-x-a
- alternatively, we may apply atom forces per rows of a grid, creating iteration order z-y-a-x
- or tiling may be used

Memory tiling

- we break some loop into nested loops, such that outer loop iterates with step $s > 1$ and those steps are performed in some inner loop
- multiple loops may be tiled
- code is more complex, not shown in this tutorial

Evaluation

Note that autovectorization is switched off for all implementations.

Implementation	Performance	speedup
Naive (grid 257)	164.7	n/a
Spatial loc.	371.8	2.26×
ZYXA	359.7	2.18×
ZYAX	382.2	2.32×

Temporal locality brings only minor improvement, but it may change when instructions are optimized/code is parallelized.

Vector Instructions

Modern processors have complex logic preparing instructions

- arithmetical units are relatively cheep
- when instruction is to be executed, it may process multiple data elements in parallel

Data-parallel programming

- the same instruction is applied onto multiple data (SIMD model)
- explicit usage: we need to generate vector instructions

Vector Instructions

Vector instructions

- the same operation is applied to a short vector
- mainly arithmetic operations, may be masked, may contain support for reduction, binning etc.
- vector length depends on data type and instruction set, e.g. AVX2 works with vector of size 256 bytes, so 8 32-bit numbers or 4 64-bit numbers are processed in parallel

Vectorization in C/C++

- explicit: inline assembly or intrinsics
- implicit: compiler generates vector instructions automatically

Automatic Vectorization

Better portability

- the code can be compiled for any vector instruction set

Supported in modern compilers

- however, it is difficult task, so allowing compiler to vectorize code needs programmer assist
- Intel C++ compiler is strong in auto-vectorization

Automatic Vectorization

Current limitations

- only innermost `for` loops are vectorized
- number of iterations must be known when loop is entered, or (preferably) at compilation time
- memory access must be regular, ideally with unit stride (i.e. consequent elements are accessed in vector instructions)
- vector dependence usually disallows vectorization

Vector Dependence

Vector dependence

- the for loop cannot be vectorized, if there is flow dependence between iterations
- however, compiler may wrongly assume vector dependence (it must by conservative to generate correct code)
- `#pragma ivdep` (Intel) or `#pragma GCC ivdep` (gcc) instruct compiler to ignore assumed vector dependences (with true dependence, compiler still do not vectorize)

Contiguous Memory Access

```
struct vec{
  float x,y;
};
vec v[n];
for (int i = 0; i < n; i++)
  v[i].x *= 2.0f;
```

The loop is vectorized, however, access into v is strided. Typical optimization is transferring array of structures (AoS) to structure of arrays (SoA).

```
struct vec{
  float *x;
  float *y;
};
vec v;
// allocation ...
for (int i = 0; i < n; i++)
  v.x[i] *= 2.0f;
```

Electrostatic Potential Map

Naive implementation has assumed dependence (compiler is not clever enough), which needs to be manually fixed.

```
for (int a = 0; a < nAtoms; a++) {
  sAtom myAtom = atoms[a];
  for (int x = 0; x < gSize; x++) {
    float dx2 = powf((float)x * gs - myAtom.x, 2.0f);
    for (int y = 0; y < gSize; y++) {
      float dy2 = powf((float)y * gs - myAtom.y, 2.0f);
      for (int z = 0; z < gSize; z++) {
        float dz = (float)z * gs - myAtom.z;
        float e = myAtom.w / sqrtf(dx2 + dy2 + dz*dz);
        #pragma ivdep
        grid[z*gSize*gSize + y*gSize + x] += e;
      }
    }
  }
}
```

Electrostatic Potential Map

AZYX and ZYAX Innermost Loop

```
for (int x = 0; x < gSize; x++) {
  float dx = (float)x * gs - myAtom.x;
  float e = myAtom.w / sqrtf(dx*dx + dy2 + dz2);
  grid[z*gSize*gSize + y*gSize + x] += e;
}
```

The loop is automatically vectorized without problems.

Electrostatic Potential Map

ZYXA implementation

```
for (int z = 0; z < gSize; z++) {
  for (int y = 0; y < gSize; y++) {
    for (int x = 0; x < gSize; x++) {
      float e = 0.0f;
      for (int a = 0; a < nAtoms; a++) {
        sAtom myAtom = atoms[a];
        float dx = (float)x * gs - myAtom.x;
        float dy = (float)y * gs - myAtom.y;
        float dz = (float)z * gs - myAtom.z;
        e += myAtom.w / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      grid[z*gSize*gSize + y*gSize + x] += e;
    }
  }
}
```

ZYXA

The innermost loop is difficult to vectorize

- strided memory access into `atoms` elements
- reduction

Two possible solutions

- AoS to SoA optimization
- vectorization of outer loop running over `x`

SoA

```
for (int z = 0; z < gSize; z++) {
  for (int y = 0; y < gSize; y++) {
    for (int x = 0; x < gSize; x++) {
      float e = 0.0f;
      for (int a = 0; a < nAtoms; a++) {
        float dx = (float)x * gs - atoms.x[a];
        float dy = (float)y * gs - atoms.y[a];
        float dz = (float)z * gs - atoms.z[a];
        e += atoms.w[a] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      grid[z*gSize*gSize + y*gSize + x] += e;
    }
  }
}
```

Outer-loop Vectorization

```
for (int z = 0; z < gSize; z++) {
  for (int y = 0; y < gSize; y++) {
    #pragma simd
    for (int x = 0; x < gSize; x++) {
      float e = 0.0f;
      for (int a = 0; a < nAtoms; a++) {
        sAtom myAtom = atoms[a];
        float dx = (float)x * gs - myAtom.x;
        float dy = (float)y * gs - myAtom.y;
        float dz = (float)z * gs - myAtom.z;
        e += myAtom.w / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      grid[z*gSize*gSize + y*gSize + x] += e;
    }
  }
}
```

All implementations

We will use `restrict` quantifier

- otherwise, compiler may expect aliasing even between `atoms` and `grid` and give up vectorization

Evaluation

Implementation	Performance	speedup	(vect. speedup)
Naive (grid 257)	164.7	n/a	n/a
Naive vec. (grid 257)	330.6	2.01×	2.01×
Spatial loc.	1838	11.2×	4.94×
ZYXA outer	2189	13.3×	6.09×
ZYXA SoA	2203	13.4×	6.12×
ZYAX	2197	13.3×	5.75×

Parallelization in C/C++

Thread-level parallelism in C/C++

- many possible ways to parallelize a code: pthreads, Boost threads, TBB etc.
- we will use OpenMP in our examples, as it broadly-supported standard and it requires only small changes in our code
- however, optimization principles are general and can be used with any parallelization interface

OpenMP

OpenMP standard

- for shared-memory parallelism
- uses pragmas to declare, which parts of the code runs in parallel
- very easy to use, but writing efficient code may be challenging (much like in other interfaces)
- implements fork-join model
- standard, implemented in all major C/C++ compilers

OpenMP

The parallel region of the code is declared by `#pragma omp parallel`

```
//serial code
const int n = 100;
#pragma omp parallel
{
  //parallel code
  printf("Hello from thread %d\n", omp_get_thread_num());
  //parallel loop, iterations order is undefined
  #pragma omp for
  for (int i = 0; i < n; i++) {
    //iteration space is distributed across all threads
    printf("%d ", i);
  }
}
//serial code
```

OpenMP

We can define private and shared variables

- `#pragma omp parallel for private(a) shared(b)`
- variables declared before parallel block are shared by default
- `private` statement creates private copy for each thread

Thread synchronization

- we can define critical section by `#pragma omp critical`
- or use lightweight atomic operations, which are restricted to simple scalar operations, such as `+ - * /`

Electrostatic Potential Map

Which loop can be parallelized?

- AZYX: loop running over atoms would need synchronization, so we prefer to parallelize loop running over Z, Y or X
- ZYXA: we can parallelize up to three outermost loops
- ZYAX: we can parallelize up to two outermost loops

Which loop to parallelize?

- enter and exit of the loop is synchronized
- we want to minimize number of synchronizations, so we will parallelize loops performing more work
- to scale better, we may collapse `n` perfectly-nested loops using `#pragma omp for collapse(n)`

ZYXA Example

```
#pragma omp parallel for
for (int z = 0; z < gSize; z++) {
  for (int y = 0; y < gSize; y++) {
    #pragma simd
    for (int x = 0; x < gSize; x++) {
      float e = 0.0f;
      for (int a = 0; a < nAtoms; a++) {
        sAtom myAtom = atoms[a];
        float dx = (float)x * gs - myAtom.x;
        float dy = (float)y * gs - myAtom.y;
        float dz = (float)z * gs - myAtom.z;
        e += myAtom.w / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      grid[z*gSize*gSize + y*gSize + x] += e;
    }
  }
}
```

ZYXA Example

```
#pragma omp parallel for collapse(2)
for (int z = 0; z < gSize; z++) {
  for (int y = 0; y < gSize; y++) {
    #pragma simd
    for (int x = 0; x < gSize; x++) {
      float e = 0.0f;
      for (int a = 0; a < nAtoms; a++) {
        sAtom myAtom = atoms[a];
        float dx = (float)x * gs - myAtom.x;
        float dy = (float)y * gs - myAtom.y;
        float dz = (float)z * gs - myAtom.z;
        e += myAtom.w / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      grid[z*gSize*gSize + y*gSize + x] += e;
    }
  }
}
```

Evaluation

Implementation	Performance	speedup	(par. speedup)
Naive (grid 257)	164.7	n/a	n/a
Spatial loc.	2272	13.8×	1.24×
ZYXA outer	7984	48.5×	3.62×
ZYAX	8092	49.1×	3.68×