Optimization study

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

D. Berényi – M. F. Nagy-Egri

Wigner GPU Lab

The C++ Programming Language

N-body simulation

We go through the optimization levels of a simple n-body simulation

N-body simulation

Wigner GPU Lab

Gravitational interaction of N point masses:

$$\vec{F} = \frac{-G M_1 M_2 (\vec{r}_1 - \vec{r}_2)}{|\vec{r}_1 - \vec{r}_2|^3}$$

This has to be summed, and later used in a simple time stepper scheme, here a forward-Euler:

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n \Delta t + \frac{1}{2} \vec{a}_n \Delta t^2$$

The C++ Programming Language

N-body simulation

Wigner GPU Lab

What should make up a particle?

```
struct particle
{
    double mass;
    std::array<double, 3> pos;
    std::array<double, 3> v;
    std::array<double, 3> f;//sum of forces on this particle
};
```

The C++ Programming Language

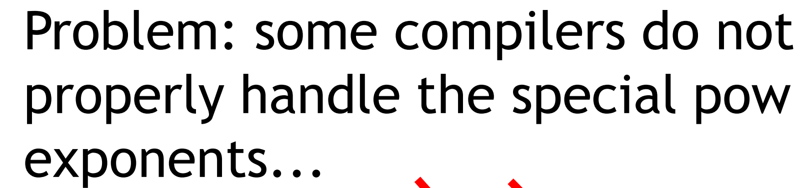
N-body simulation

For the force we need the distance.
A naïve implementation:

```
particle p1, p2;
double d = std::pow( std::pow(p1.pos[0] - p2.pos[0], 2) +
                     std::pow(p1.pos[1] - p2.pos[1], 2) +
                     std::pow(p1.pos[2] - p2.pos[2], 2), 0.5);
```

N-body simulation

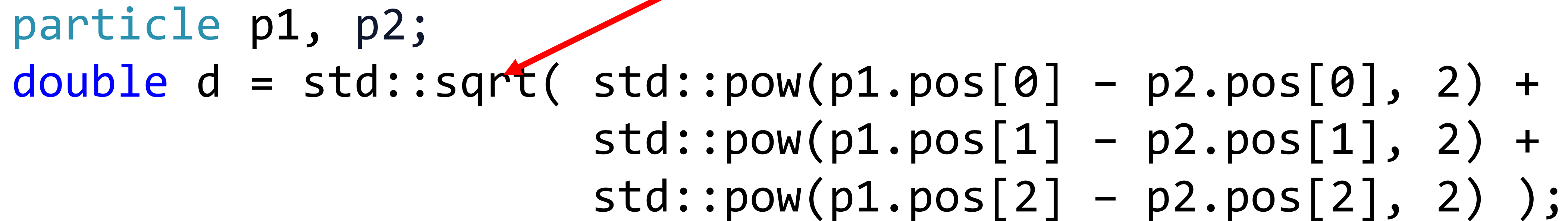
For the force we need the distance.
A naïve implementation:




```
particle p1, p2;
double d = std::pow( std::pow(p1.pos[0] - p2.pos[0], 2) +
                     std::pow(p1.pos[1] - p2.pos[1], 2) +
                     std::pow(p1.pos[2] - p2.pos[2], 2), 0.5);
```

N-body simulation

For the force we need the distance.
A better implementation: be specific!

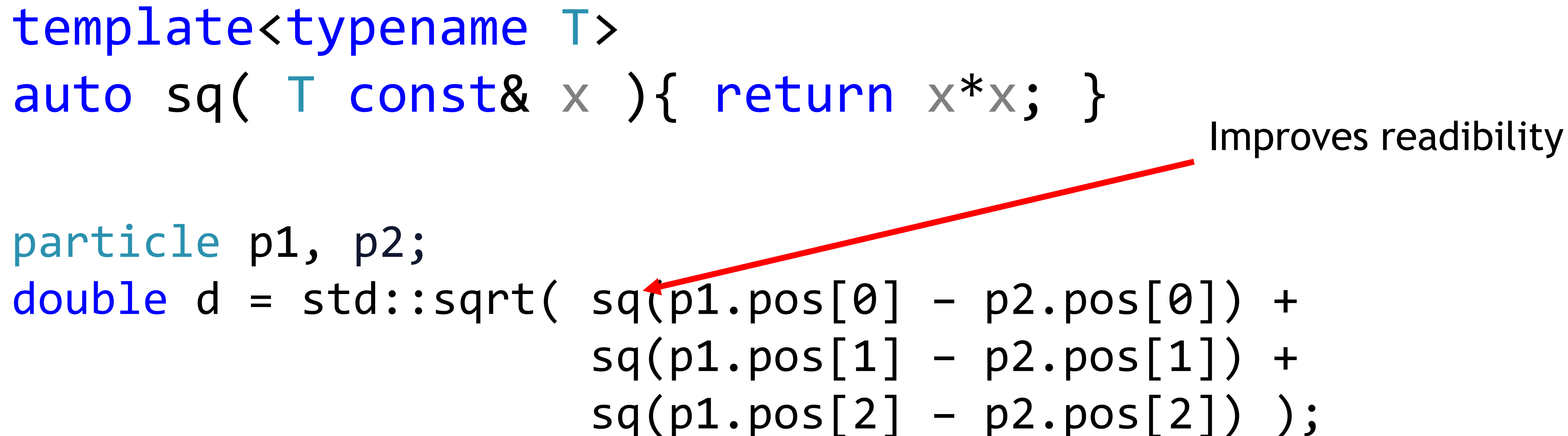


```
particle p1, p2;
double d = std::sqrt( std::pow(p1.pos[0] – p2.pos[0], 2) +
                      std::pow(p1.pos[1] – p2.pos[1], 2) +
                      std::pow(p1.pos[2] – p2.pos[2], 2) );
```

N-body simulation

For the force we need the distance.
A better implementation:

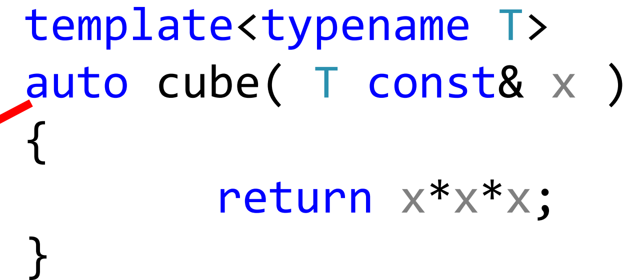
A helper function:




```
template<typename T>
auto sq( T const& x ){ return x*x; }

particle p1, p2;
double d = std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) );
```

N-body simulation

To be more precise we need: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$




```
particle p1, p2;

double fx =
(p1.pos[0] - p2.pos[0]) / cube(
            std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                       sq(p1.pos[1] - p2.pos[1]) +
                       sq(p1.pos[2] - p2.pos[2]) ) );

double fy =
(p1.pos[1] - p2.pos[1]) / cube(
            std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                       sq(p1.pos[1] - p2.pos[1]) +
                       sq(p1.pos[2] - p2.pos[2]) ) );
```

N-body simulation

Wigner GPU Lab

To be more precise we need: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

It is good to factor out as much as possible:

```
particle p1, p2;

double factor = cube(
           std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) ) );
double fx = (p1.pos[0] - p2.pos[0]) / factor;
double fy = (p1.pos[1] - p2.pos[1]) / factor;
double fz = (p1.pos[2] - p2.pos[2]) / factor;
```

The C++ Programming Language

N-body simulation

Wigner GPU Lab

To be more precise we need: $\frac{\vec{r}_1-\vec{r}_2}{|\vec{r}_1-\vec{r}_2|^3}$

It is good to factor out as much as possible:

```
particle p1, p2;

double dx = p1.pos[0] - p2.pos[0];
double dy = p1.pos[1] - p2.pos[1];
double dz = p1.pos[2] - p2.pos[2];
double factor = cube( std::sqrt( sq(dx) + sq(dy) + sq(dz) ) );
double fx = dx / factor;
double fy = dy / factor;
double fz = dx / factor;
```

The C++ Programming Language

N-body simulation

WIGNER GPU Lab

To be more precise we need: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

Division is an expensive operation, lets carry out it only once:

*notes in order

```
particle p1, p2;

double dx = p1.pos[0] – p2.pos[0];
double dy = p1.pos[1] – p2.pos[1];
double dz = p1.pos[2] – p2.pos[2];
double factor =
1.0 / cube( std::sqrt( sq(dx) + sq(dy) + sq(dz) ) );
double fx = dx * factor;
double fy = dy * factor;
double fz = dx * factor;
```

THE C++ PROGRAMMING LANGUAGE

N-body simulation

The full force calculation will look like:

```
particle p1, p2;
double dx = p1.pos[0] – p2.pos[0];
double dy = p1.pos[1] – p2.pos[1];
double dz = p1.pos[2] – p2.pos[2];
double factor =
1.0 / cube( std::sqrt( sq(dx) + sq(dy) + sq(dz) ) );
double fx = -G * p1.mass * p2.mass * dx * factor;
double fy = -G * p1.mass * p2.mass * dy * factor;
double fz = -G * p1.mass * p2.mass * dz * factor;
```

Wigner GPU Lab

The C++ Programming Language

N-body simulation

The full force calculation will look like:

Better:

```
particle p1, p2;

double dx = p1.pos[0] – p2.pos[0];
double dy = p1.pos[1] – p2.pos[1];
double dz = p1.pos[2] – p2.pos[2];
double factor =
-G * p1.mass * p2.mass /
                cube( std::sqrt( sq(dx) + sq(dy) + sq(dz) ) );
double fx = dx * factor;
double fy = dy * factor;
double fz = dz * factor;
```

Note:
Now all fields of the particle struct is read only once!

N-body simulation

Also, the final code for time stepping will look like:

```
void forward_euler(particle& part, const double& dt){
   auto dt_per_m = dt / part.mass; auto dt_sq = dt*dt;
   auto half_dt_sq_per_m = 0.5 * dt_sq / part.mass;

   part.pos[0] += part.v[0] * dt + part.f[0] * half_dt_sq_per_m;
   part.pos[1] += part.v[1] * dt + part.f[1] * half_dt_sq_per_m;
   part.pos[2] += part.v[2] * dt + part.f[2] * half_dt_sq_per_m;

   part.v[0] += part.f[0] * dt_per_m;
   part.v[1] += part.f[1] * dt_per_m;
   part.v[2] += part.f[2] * dt_per_m;
   part.f = { 0.0, 0.0, 0.0 };
}
```

N-body simulation

Wigner GPU Lab

Notes:

- These refactorings may be carried out by the optimizing compiler, but may be not.
  Writing it out makes it more direct
  (you still can't be certain in some cases)

- These refactorings change the rounding semantics.
  The loss of significant bits may not be the same in the two cases.
  You need to have an idea of the magnitudes of your numbers!

The C++ Programming Language

N-body simulation

Wigner GPU Lab

Final code:

```
std::array<double, 3> calculate_force(const particle& p1,
const particle& p2)
{
   std::array<double, 3> dr
   { p1.pos[0] - p2.pos[0], p1.pos[1] - p2.pos[1],
     p1.pos[2] - p2.pos[2] };

   double d = std::sqrt( sq(dr[0]) + sq(dr[1]) + sq(dr[2]) );
   double f = -G * p1.mass * p2.mass / cube(d);
   return { f * dr[0], f * dr[1], f * dr[2] };
}
```

The C++ Programming Language

N-body simulation

To sum up all the forces, we need to iterate the force calculation on all particles

Assume we have a

```
std::vector< particle > particles;
```

N-body simulation

Wigner GPU Lab

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != particles.cend(); ++it)
  {
    if (IT != it)//no self interaction
    {
      auto f = calculate_force(*IT, *it);
      force[0] += f[0];
      force[1] += f[1];
      force[2] += f[2];
    }

  }
  IT->f = force;
}
```

Better not accumulate directly to the particle object! Use a temporary!

The C++ Programming Language

N-body simulation

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != particles.cend(); ++it)
  {
    if (IT != it)//no self interaction
    {
      auto f = calculate_force(*IT, *it);
      force[0] += f[0];
      force[1] += f[1];
      force[2] += f[2];
    }

  }
  IT->f = force;
}
```

Do not write conditionals into loops if it is avoidable!!!

N-body simulation

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT){
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != IT; ++it)
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  for(auto it = std::next(IT); it != particles.cend(); ++it )
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  IT->f = force; }
```

We broke the loop into two parts, skipping the self interaction!

N-body simulation

Lets do some caching!

The inner loop must not run from begin to end on every iteration of the outer: we create blocks of it,
thus it remains in the cache!

We restrict our working set of particles to this block

N-body simulation

```
size_t N = 1024;//block size
for( auto cache = particles.begin(); cache != particles.end(); cache += N ){
  for( auto IT = particles.begin(); IT != particles.end(); ++IT ){
    auto before = intersect( particles.begin(), IT, cache, cache + N );
    auto after  = intersect( std::next( IT ), particles.end(), cache, cache + N );
    std::array<double, 3> force{ 0.0, 0.0, 0.0 };

    for( auto it = before.first; it != before.second; ++it )
    { /*calculate_force( *IT, *it ); accumulate to force*/ }
    for ( auto it = after.first; it != after.second; ++it )
    { /*calculate_force( *IT, *it ); accumulate to force*/ }

    IT->f[0] += force[0]; IT->f[1] += force[1]; IT->f[2] += force[2];
  }
}
```

We split the ranges at IT!

Wigner GPU Lab

The C++ Programming Language

N-body simulation

The intersect helper function:

```
template <typename RndIt>
std::pair<RndIt, RndIt> mask_range( RndIt first1, RndIt last1,
                                     RndIt first2, RndIt last2 )
{
   if ( ((last2 < first1) || (last1 < first2)) )
   {
      return std::make_pair( first1, first1 );
   }
   else{
     return std::make_pair( std::max( first1, first2 ),
                            std::min( last1 , last2  )  ); }
}
```

N-body simulation

So far, so good, but we doing 2 times the work needed, since the force calculation is symmetric.

N-body simulation

```
size_t N = 1024;//block size
for( auto cache =  particles.begin();
          cache != particles.end(); cache += N ){

   for( auto IT = particles.begin(); IT != particles.end(); ++IT ){

     auto before = intersect( particles.begin(), IT, cache, cache + N );
      for( auto it = before.first; it != before.second; ++it )
      {
        auto force = calculate_force( *IT, *it );
        IT->f[0] += force[0]; IT->f[1] += force[1]; IT->f[2] += force[2];
        it->f[0] -= force[0]; it->f[1] -= force[1]; it->f[2] -= force[2];
      }
   }
}
```

N-body simulation

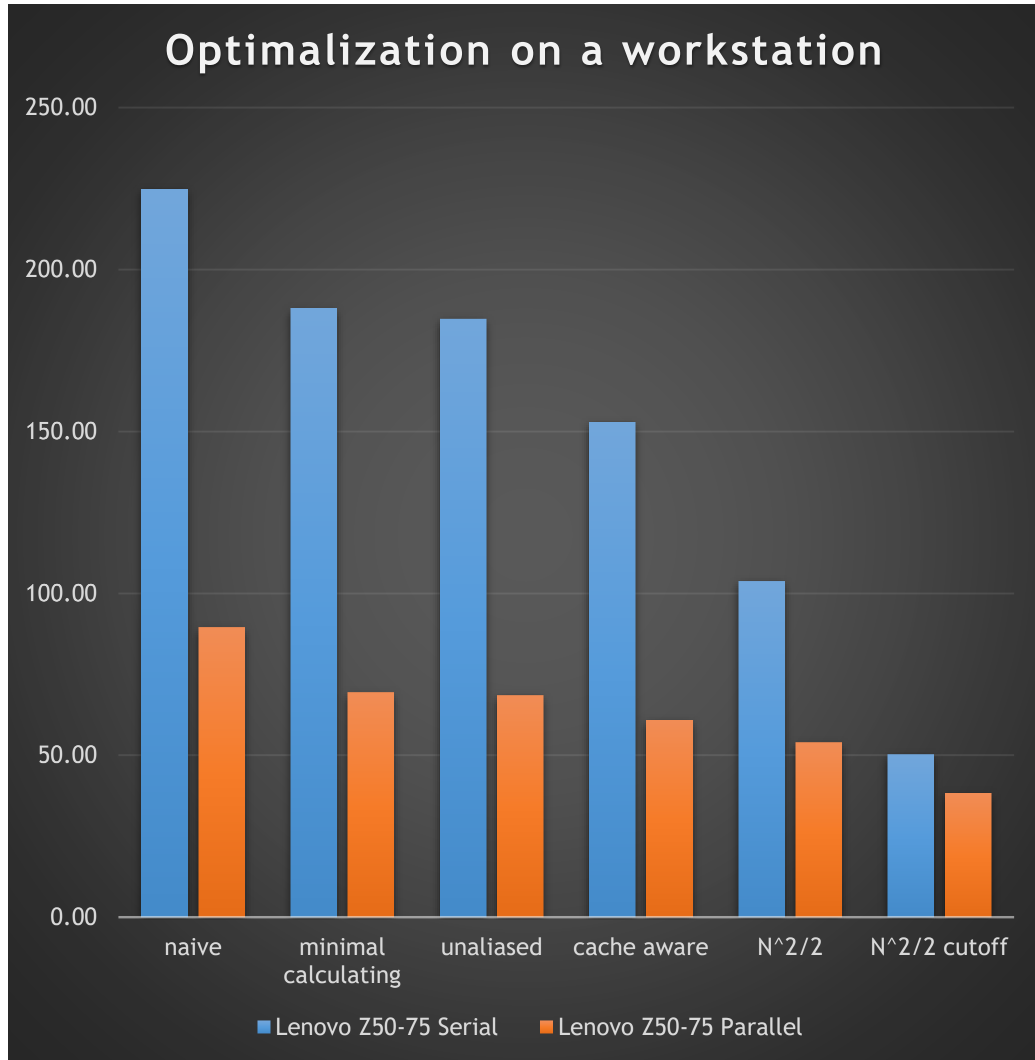
How much does this mean in terms of time?

N-body simulation (by CTest)

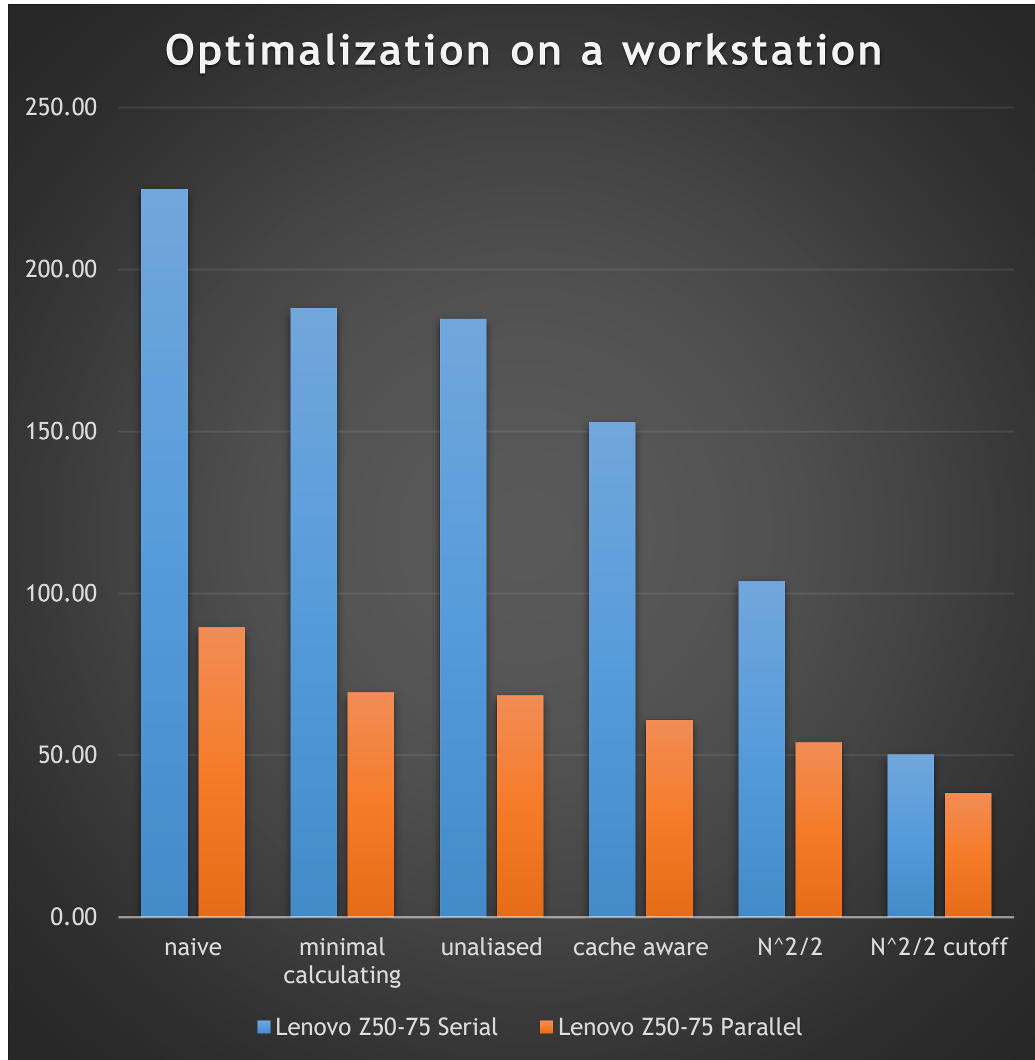
```
PS C:\Users\Matty\Build\comp-phys\NMake> ctest
Test project C:/Users/Matty/Build/comp-phys/NMake
      Start  1: manybody-c++-v1-serial
 1/12 Test  #1: manybody-c++-v1-serial ...........   Passed   58.70 sec
      Start  2: manybody-c++-v1-parallel
 2/12 Test  #2: manybody-c++-v1-parallel .........   Passed   29.71 sec
      Start  3: manybody-c++-v2-serial

…

11/12 Test #11: manybody-c++-v6-serial ...........   Passed   19.83 sec
      Start 12: manybody-c++-v6-parallel
12/12 Test #12: manybody-c++-v6-parallel .........   Passed   16.15 sec

100% tests passed, 0 tests failed out of 12

Total Test time (real) = 415.55 sec
```

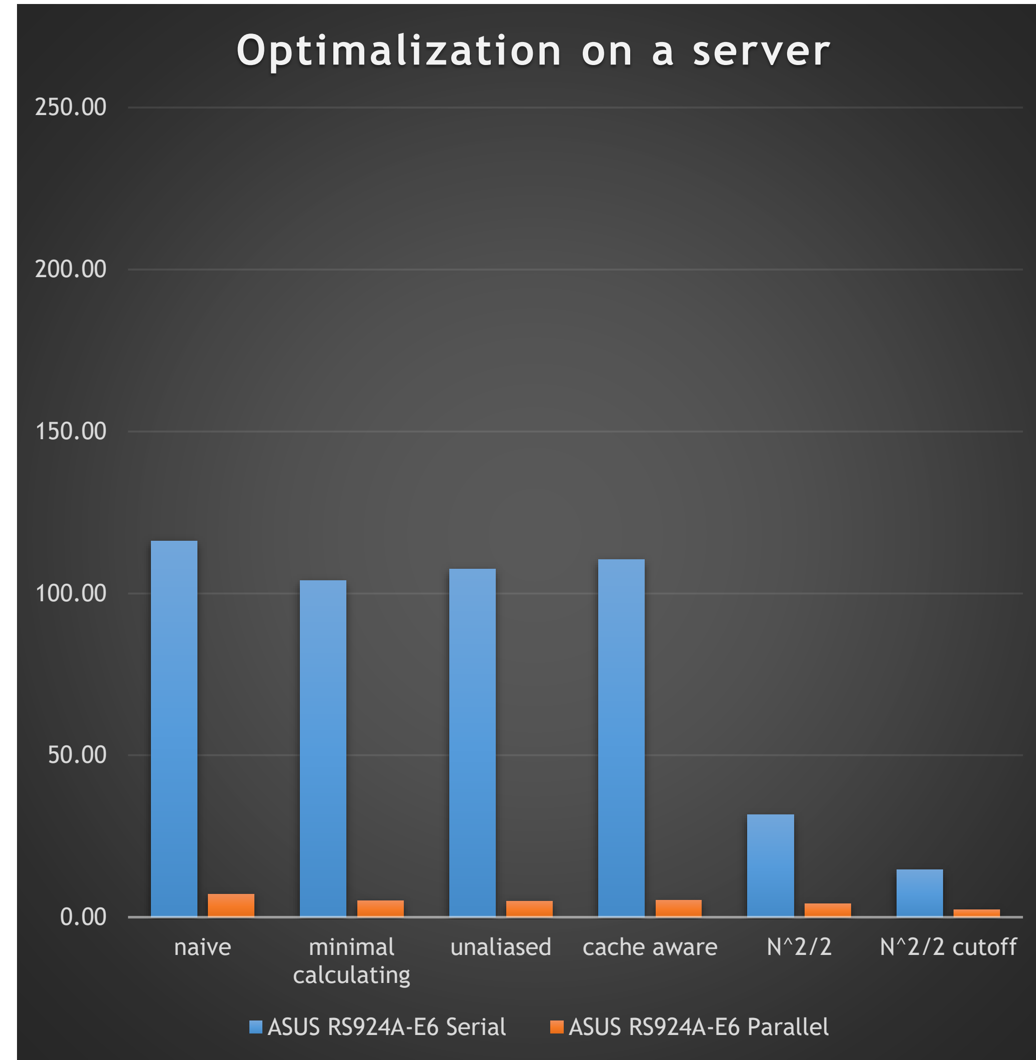

The C++ Programming Language

N-body simulation (64K particles)



N-body simulation (64K particles)





N-body simulation (64K particles)

Wigner GPU Lab

Optimization level	Lenovo Z50-75		ASUS RS924A-E6	
	Serial	Parallel (4 threads)	Serial	Parallel (64 threads)
naive	225s	89s	116s	7s
refactored	188s	69s	104s	5s
unaliased	185s	69s	108s	5s
cache aware	153s	61s	110s	5s
N^2/2	104s	54s	32s	4s

The C++ Programming Language

N-body simulation

Other possible algorithmic optimizations in the N-body simulation:

- Cut-off distance
- Space partitioning

Common optimization patterns

Some other common optimization patterns:

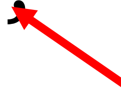
Common optimization patterns

Some other common optimization patterns:

- Order the fields in the struct based on access frequency

```
struct particle
{
    std::array<double, 3> pos;
    std::array<double, 3> f;
    std::array<double, 3> v;
    double mass;
};
```

Most often needed



Less often needed

Common optimization patterns

Some other common optimization patterns:

- Do not mix different sized types, especially small types
  Place larger fields first

BAD

```
struct particle
{
    std::array<double, 3> pos;
    bool bActive
    std::array<double, 3> v;
    double mass;
};
```

Better

```
struct particle
{
    std::array<double, 3> pos;
    std::array<double, 3> v;
    double mass;
    bool bActive
};
```

The C++ Programming Language

Common optimization patterns

Wigner GPU Lab

Some other common optimization patterns:

- For many sequential access patterns
  Structures of Arrays are better than Arrays of Structures

```
std::vector< particle > particles;

std::vector< std::array<double, 3> > part_pos;
std::vector< std::array<double, 3> > part_v;
std::vector< std::array<double, 3> > part_f;
std::vector< double > part_mass;
```

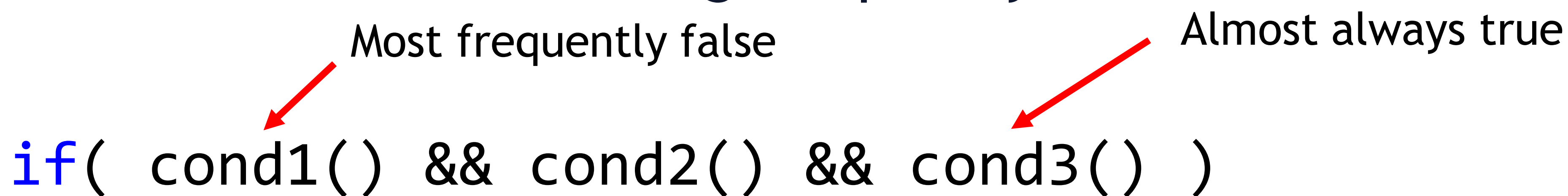
In some cases these are better

The C++ Programming Language

Common optimization patterns

Some other common optimization patterns:

- If you have multiple and-ed expensive conditionals, but you know how often the predicates are true, order them in ascending frequency:




```
if( cond1() && cond2() && cond3() )
```

This works because C++ is lazy: it will not evaluate cond2() and cond3() if the first one fails!