Cloth Simulation Optimisation Report

Will (Bogong) Wang

1. Initial Performance Assessment

Motivation: gather performance data for different problem sizes as a baseline for future comparison.

After conducting an analysis of the source file `cloth_code_main.cpp`, we can deduce that the computational complexity of the function loop_code is given by $\mathcal{O}\left(n^2(2d+1)^2 + 6n^2\right) = \mathcal{O}(n^2d^2)$, where n represents the number of nodes per dimension and d signifies the level of node interaction. Subsequently, we evaluate the performance of the function kernel_main under varying conditions of n and d.

The overall complexity of the cloth simulation algorithm can be expressed as $\mathcal{O}(n^2d^2i)$, taking into account that the simulation must be executed for i iterations.

For the experimental evaluation, the default settings are adopted as follows: $s = 1.0$, $m = 1.0$, $f = 10.0$, $g = 0.981$, $b = 3.0$, $o = 0.0$, $t = 0.05$, and $i = 100$. Performance metrics are accumulated under varied n and d, and are tabulated in reference Table 1. The nomenclature for column headers is elucidated in Table 12.

Figure 1 reveals that the observed program complexity, quantified in terms of wall time, aligns closely with the anticipated complexity.

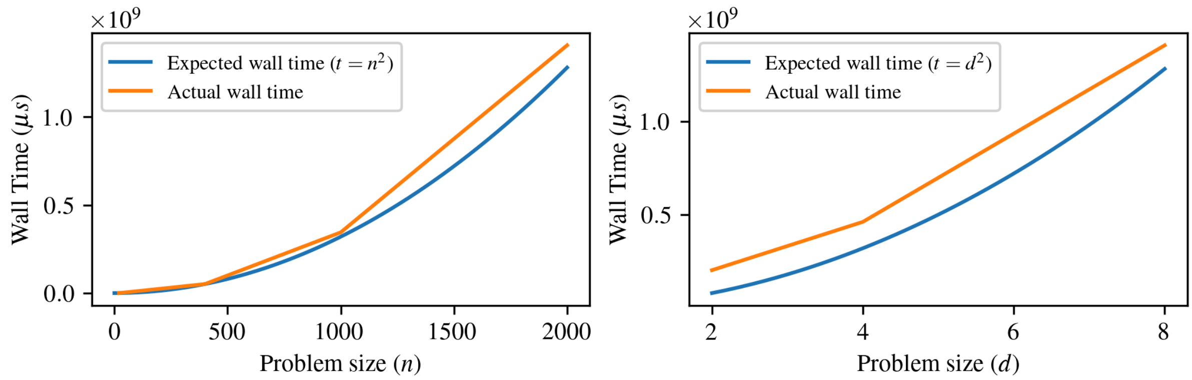


Figure 1: Expected performance vs. actual performance for different problem sizes

n	d	wall time (μs)	PAPI_DP_OPS	MFLOPS
20	2	12,179	22,089,704	1813
20	4	23,378	62,313,792	2,665
20	8	55,758	173,229,236	3,109
400	2	5,422,059	9,878,527,384	1,821
400	4	15,839,120	31,095,399,360	1,963
400	8	51,357,925	108,974,382,748	2,121
1000	2	43,575,648	61,956,127,384	1,421

1000	4	106,468,567	195,636,999,360	1,837
1000	8	344,724,516	689,837,263,908	2,001
2000	2	202,672,404	248,112,127,384	1,224
2000	4	461,251,560	784,272,999,360	1,700
2000	8	1,406,303,685	2,771,062,063,908	1,970

PAPI_L1_DCM	**PAPI_L2_DCM**	**PAPI_TOT_INS**	**PAPI_BR_MSP**	**PAPI_VEC_DP**
74,024	968	59,412,923	6,520	23,588,148
72,744	898	158,120,942	20,537	67,167,004
78,392	1,994	426,922,289	92,320	187,331,832
635,860,924	78,514,287	25,930,771,500	291,394	10,569,726,308
704,238,219	79,158,784	77,843,410,225	335,098	33,554,671,420
1,136,689,807	93,105,002	265,829,781,566	31,980,462	117,923,575,376
4,396,781,389	547,209,567	162,580,080,170	732,867	66,294,126,308
4,560,247,819	624,216,570	489,638,592,689	880,648	211,115,071,420
4,926,933,755	835,299,429	1,682,441,123,754	199,971,575	746,498,696,296
21,111,961,856	13,525,293,251	651,009,015,227	1,522,236	265,488,126,308
21,756,632,703	14,419,459,456	1,962,723,938,221	1,982,042	846,329,071,420
24,442,697,761	17,549,567,396	6,757,899,995,171	800,306,253	2,998,683,896,296

Table 1: Performance data of `kernel_main` for different problem sizes

2. Serial Code Optimisation

Motivation: optimise the serial code before parallelisation.

2.1. Memory Access Optimisation

2.1.1. Optimise memory access pattern and cache locality

Optimising memory access patterns enhances the spatial locality within the cache, thereby contributing to improved memory performance. An example is illustrated below to substantiate this claim. A comparative analysis reveals that prior to optimisation, memory is accessed with a stride of n. After optimisation, the stride is reduced to 1, thereby making memory access more efficient.

```
before
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    x[j * n + i] += dt * (vx[j * n + i] + dt * fx[j * n + i] * 0.5 / mass);
    oldfx[j * n + i] = fx[j * n + i];
  }
}
...
```

```
after
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    x[i * n + j] += dt * (vx[i * n + j] + dt * fx[i * n + j] * 0.5 / mass);
    oldfx[i * n + j] = fx[i * n + j];
  }
}
...
```

Another optimisation is focusing on temporal locality, which aims to keep frequently or recently accessed data in cache memory for quicker retrieval. An example is shown below.

```
before
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    x[i * n + j] += dt * (vx[i * n + j] + dt * fx[i * n + j] * 0.5 / mass);
    ...
  }
}
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    y[i * n + j] += dt * (vy[i * n + j] + dt * fy[i * n + j] * 0.5 / mass);
    ...
  }
}
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    z[i * n + j] += dt * (vz[i * n + j] + dt * fz[i * n + j] * 0.5 / mass);
    ...
```

```
  }
}
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    xdiff = x[i * n + j] - xball;
    ydiff = y[i * n + j] - yball;
    zdiff = z[i * n + j] - zball;
    ...
  }
}
```

```
after
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    x[i * n + j] += dt * (vx[i * n + j] + dt * fx[i * n + j] * 0.5 / mass);
    ...
    y[i * n + j] += dt * (vy[i * n + j] + dt * fy[i * n + j] * 0.5 / mass);
    ...
    z[i * n + j] += dt * (vz[i * n + j] + dt * fz[i * n + j] * 0.5 / mass);
    ...
    xdiff = x[i * n + j] - xball;
    ydiff = y[i * n + j] - yball;
    zdiff = z[i * n + j] - zball;
    ...
  }
}
...
```

2.1.2. Reduce memory access

Memory access is inherently time-intensive; therefore, the second optimisation aims to minimise the frequency of both read and write operations. This can be achieved through the frequent use of registers to obviate redundant memory accesses. Accessing data from registers is typically the fastest method, often requiring only a single cycle. Temporary variables can be created and employed for this purpose, serving as in-register storage. A comparative example showing this optimisation is presented below.

```
before
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    xdiff = x[i * n + j] - ...;
    ...
    if (vmag < rball) {
      ...
      x[i * n + j] = xball + ...;
      ...
    }
  }
}
...
```

```
after
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    double x_tmp = x[i * n + j];
    xdiff = x_tmp - ...;
    ...
    if (vmag < rball) {
      ...
      x_tmp = xball + ...;
      ...
    }
  }
}
...
```

2.1.3. Results

Subsequent to the implementation of memory access optimisation, performance metrics were re-acquired and are delineated in Figure 2 and Table 2. The data indicates a notable enhancement in performance, resulting a 14% reduction in wall time. Moreover, a significant cache performance also shows. The optimised code has decreases in L1 and L2 cache misses was observed, registering reductions of 64% and 51%, respectively.

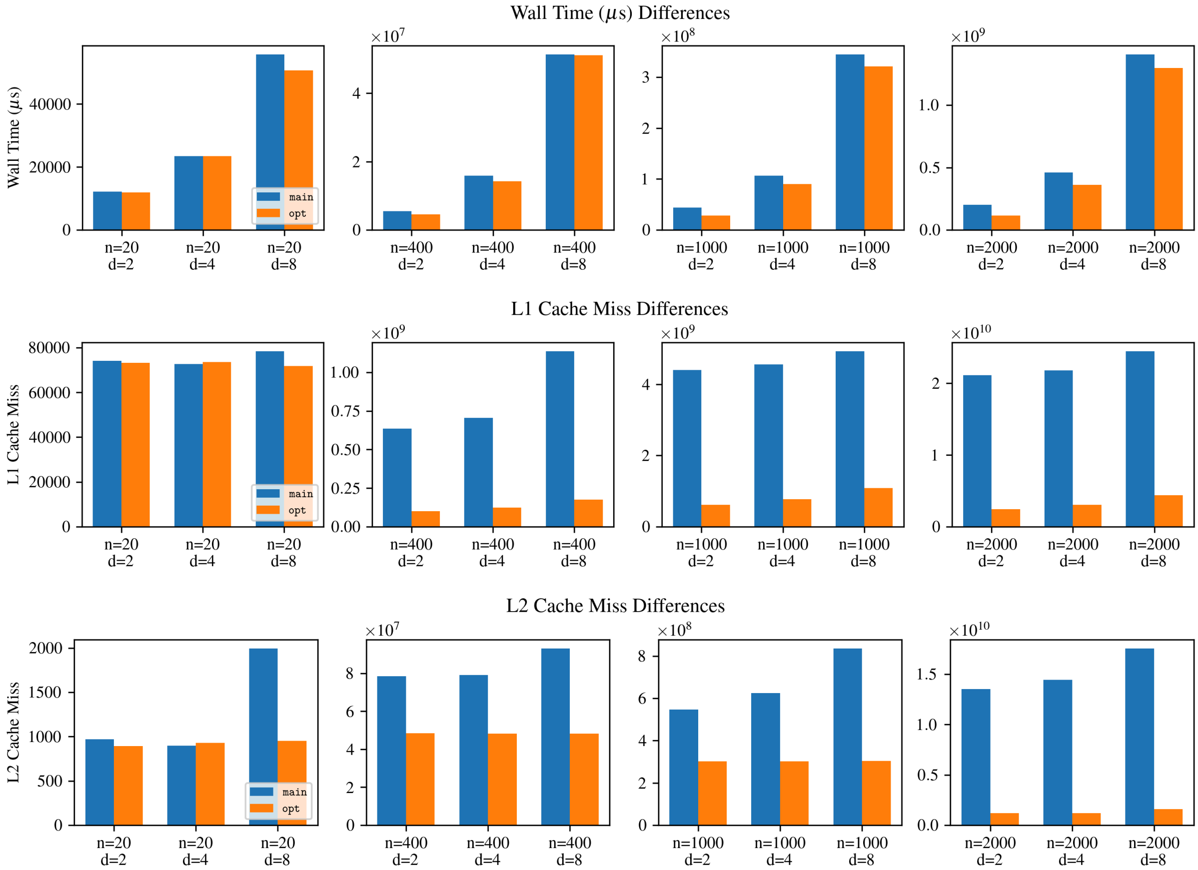


Figure 2: Performance comparison between `kernel_main` and `kernel_opt` after memory access optimisation

n	d	wall time (μs)	`PAPI_DP_OPS`	`MFLOPS`
20	2	11,896	22,080,948	1,856
20	4	23,497	62,307,004	2,651
20	8	50,706	173,227,032	3,054
400	2	4,524,730	9,878,519,108	2,183
400	4	14,218,072	31,095,391,420	2,187
400	8	51,110,788	108,974,378,576	2,132
1000	2	28,521,728	61,956,119,108	2,172
1000	4	89,820,996	195,636,991,420	2,178
1000	8	321,185,042	689,837,259,496	2,147
2000	2	114,044,293	248,112,119,108	2,175

2000	4	362,111,465	784,272,991,420	2,165
2000	8	1,297,440,123	2,771,062,059,496	2,135

PAPI_L1_DCM	PAPI_L2_DCM	PAPI_TOT_INS	PAPI_BR_MSP	PAPI_VEC_DP
73,197	894	53,044,857	7,560	22,080,948
73,544	929	137,106,796	23,253	62,307,004
71,816	953	363,154,637	245,354	173,227,032
99,396,813	48,339,202	23,050,236,882	153,301	9,878,519,108
125,128,172	48,305,650	67,137,550,757	301,732	31,095,391,420
176,370,035	48,316,791	224,689,104,012	50,831,354	108,974,378,576
615,634,361	301,281,856	144,503,909,192	386,861	61,956,119,108
770,970,490	301,367,420	422,239,623,631	762,453	195,636,991,420
1,082,951,754	303,082,738	1,421,842,187,273	306,899,531	689,837,259,496
2,436,555,978	1,203,770,857	578,606,720,002	793,123	248,112,119,108
3,064,317,505	1,212,944,047	1,692,476,434,126	1,870,266	784,272,991,420
4,412,580,666	1,588,852,789	5,710,857,602,605	1,215,732,179	2,771,062,059,496

Table 2: Full performance data of `kernel_opt` after memory access optimisation

2.2. Computational Optimisation

2.2.1. Avoid redundant computations

Reduce redundant computations can directly reduce the number of instructions for calculations, hence increases the speed of the program. An example of such optimisation is shown below.

before

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    x[i * n + j] += dt * (vx[i * n + j] + dt * fx[i * n + j] * 0.5 / mass);
    ...
    y[i * n + j] += dt * (vy[i * n + j] + dt * fy[i * n + j] * 0.5 / mass);
    ...
    z[i * n + j] += dt * (vz[i * n + j] + dt * fz[i * n + j] * 0.5 / mass);
    ...
  }
}
...
```

after

```
double half_dt_div_mass = dt * 0.5 / mass;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    x[i * n + j] += dt * (vx[i * n + j] + fx[i * n + j] * half_dt_dv_mass);
    ...
    y[i * n + j] += dt * (vy[i * n + j] + fy[i * n + j] * half_dt_dv_mass);
    ...
    z[i * n + j] += dt * (vz[i * n + j] + fz[i * n + j] * half_dt_dv_mass);
```

```
    ...
  }
}
...
```

2.2.2. Remove branches

Reducing branches improves CPU pipelining by making instruction flow more predictable. It also enables better vectorisation by allowing for efficient use of single instruction multiple data (SIMD) operations. There are generally two ways of removing branches in a loop, reorganising the loop structure and masking.

Below example shows how reorganising work. In our code, we avoid the branches by separating the loop body into 4 parts, top stride, middle-left stride, middle-right stride and bottom stride.

before

```
for (ny = 0; ny < n; ny++) {
  for (nx = 0; nx < n; nx++) {
    ...
    // loop over displacements
    for (dy = MAX(ny - delta, 0); dy < MIN(ny + delta + 1, n); dy++) {
      for (dx = MAX(nx - delta, 0); dx < MIN(nx + delta + 1, n); dx++) {
        // exclude self interaction
        if (nx != dx || ny != dy) {
          ...
        }
      }
    }
  }
}
```

after

```
for (ny = 0; ny < n; ny++) {
  for (nx = 0; nx < n; nx++) {
    ...
    dx_start = MAX(nx - delta, 0), dx_end = MIN(nx + delta + 1, n);
    dy_start = MAX(ny - delta, 0), dy_end = MIN(ny + delta + 1, n);
    // Top stride
    for (dx = dx_start; dx < nx; dx++) {
      for (dy = dy_start; dy < dy_end; dy++) {
        ...
      }
    }

    // Middle strides
    dx = nx;
    for (dy = dy_start; dy < ny; dy++) {
      ...
    }
    for (dy = ny + 1; dy < dy_end; dy++) {
      ...
    }
```

```
    // Bottom stride
    for (dx = nx + 1; dx < dx_end; dx++) {
      for (dy = dy_start; dy < dy_end; dy++) {
        ...
      }
    }
  }
}
```

Masking optimisation was initially explored, however, it was ultimately omitted due to its computational inefficiency. As evidenced by the subsequent example, the removal of branches paradoxically increased the number of floating-point operations, negating the intended benefits. Consequently, this optimisation was not incorporated into the final optimisation.

```
before
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    ...
    vmag = sqrt(xdiff * xdiff + ydiff * ydiff + zdiff * zdiff);
    if (vmag < rball) {
      inv_vmag = 1 / (vmag);
      ...
      x_tmp = xball + xdiff_unit * rball;
      y_tmp = yball + ydiff_unit * rball;
      z_tmp = zball + zdiff_unit * rball;
      ...
      *x_vel = 0.1 * (*x_vel - xdiff_unit * proj_scalar);
      *y_vel = 0.1 * (*y_vel - ydiff_unit * proj_scalar);
      *z_vel = 0.1 * (*z_vel - zdiff_unit * proj_scalar);
    }
    ...
  }
}
```

```
after
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    ...
    vmag = sqrt(xdiff * xdiff + ydiff * ydiff + zdiff * zdiff);
    double mask = vmag < rball;
    double inv_mask = vmag >= rball;
    inv_vmag = mask / (vmag + inv_mask);
    ...
    x_tmp = xball + xdiff_unit * rball * mask + x_tmp * inv_mask;
    y_tmp = yball + ydiff_unit * rball * mask + y_tmp * inv_mask;
    z_tmp = zball + zdiff_unit * rball * mask + z_tmp * inv_mask;
    ...
    proj_scalar = (*x_vel * xdiff_unit + *y_vel * ydiff_unit + *z_vel * zdiff_unit);
    *x_vel = 0.1 * (*x_vel - xdiff_unit * proj_scalar) * mask + *x_vel * inv_mask;
    *y_vel = 0.1 * (*y_vel - ydiff_unit * proj_scalar) * mask + *y_vel * inv_mask;
    *z_vel = 0.1 * (*z_vel - zdiff_unit * proj_scalar) * mask + *z_vel * inv_mask;
    ...
```

```
  }
}
```

2.2.3. Use lookup table

Using a lookup table achieves rapid data retrieval and enhances computational efficiency by eliminating the need for redundant calculations. In our case, we can observe that the reference distance between two nodes can be reused. The optimisation is as follows.

```
before
for (ny = 0; ny < n; ny++) {
  for (nx = 0; nx < n; nx++) {
    ...
    // Top stride
    for (dx = dx_start; dx < nx; dx++) {
      for (dy = dy_start; dy < dy_end; dy++) {
        rlen = sqrt((double)((nx - dx) * (nx - dx) + (ny - dy) * (ny - dy))) * sep;
        ...
      }
    }

    // Middle strides
    dx = nx;
    for (dy = dy_start; dy < ny; dy++) {
      rlen = sqrt((double)((nx - dx) * (nx - dx) + (ny - dy) * (ny - dy))) * sep;
      ...
    }
    for (dy = ny + 1; dy < dy_end; dy++) {
      rlen = sqrt((double)((nx - dx) * (nx - dx) + (ny - dy) * (ny - dy))) * sep;
      ...
    }

    // Bottom stride
    for (dx = nx + 1; dx < dx_end; dx++) {
      for (dy = dy_start; dy < dy_end; dy++) {
        rlen = sqrt((double)((nx - dx) * (nx - dx) + (ny - dy) * (ny - dy))) * sep;
        ...
      }
    }
  }
}
```

```
after
// Pre-compute reference distance
int size = (2 * delta + 1);
for (dx = -delta; dx <= delta; dx++) {
  for (dy = -delta; dy <= delta; dy++) {
    loop_idx = (dx + delta) * size + (dy + delta);
    rlen_table[loop_idx] = sqrt((double) (dx * dx + dy * dy)) * sep;
  }
}

for (ny = 0; ny < n; ny++) {
```

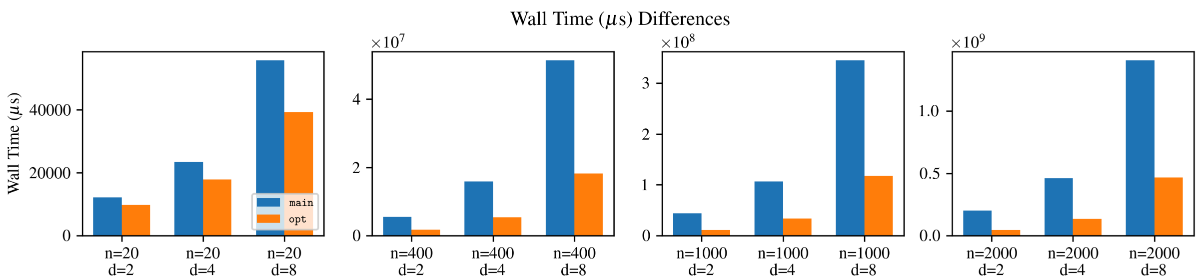
```
    for (nx = 0; nx < n; nx++) {
      ...
      // Top stride
      for (dx = dx_start; dx < nx; dx++) {
        for (dy = dy_start; dy < dy_end; dy++) {
          rlen = rlen_table[(dx - nx + delta) * size + (dy -ny + delta)];
          ...
        }
      }

      // Middle strides
      dx = nx;
      for (dy = dy_start; dy < ny; dy++) {
        rlen = rlen_table[(dx - nx + delta) * size + (dy -ny + delta)];
        ...
      }
      for (dy = ny + 1; dy < dy_end; dy++) {
        rlen = rlen_table[(dx - nx + delta) * size + (dy -ny + delta)];
        ...
      }

      // Bottom stride
      for (dx = nx + 1; dx < dx_end; dx++) {
        for (dy = dy_start; dy < dy_end; dy++) {
          rlen = rlen_table[(dx - nx + delta) * size + (dy -ny + delta)];
          ...
        }
      }
    }
  }
```

2.2.4. Results

After conducting computational optimisation, performance metrics were re-evaluated and are presented in Table 3 and Figure 3. The data reveal a significant enhancement in performance, with a 52% increase following memory access optimisation and a 58% improvement relative to the baseline metrics. Additionally, a 9% reduction in total floating-point operations was observed, which correlates with a 37% decline in the overall instruction count. The branch misprediction rate also exhibited a notable reduction, averaging a 20% decrease.



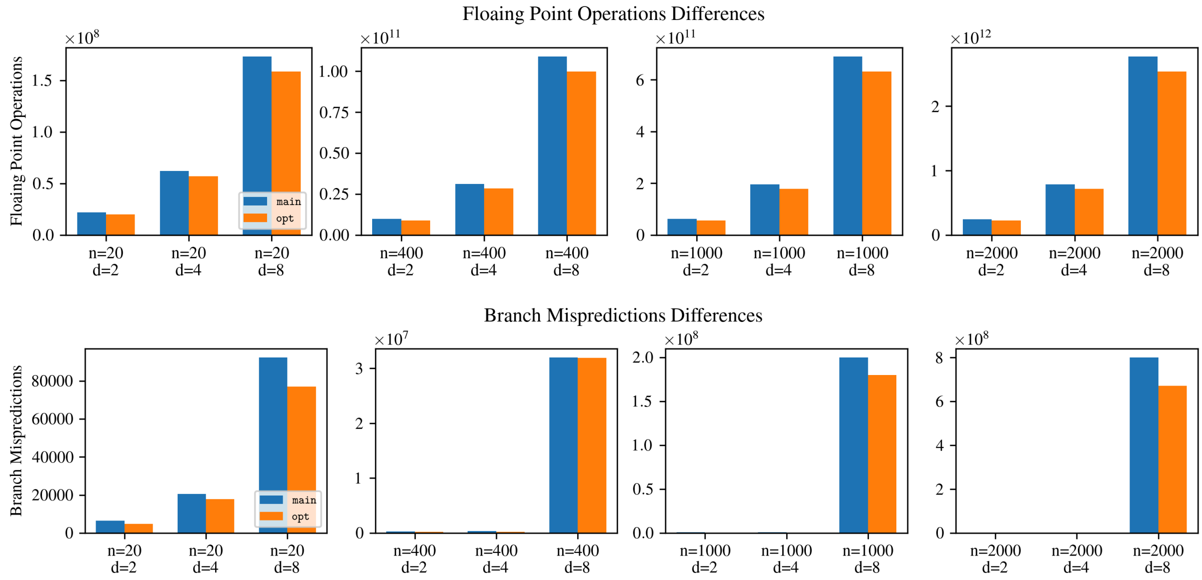


Figure 3: Performance comparison between `kernel_main` and `kernel_opt` after computational optimisation

n	d	wall time (μs)	`PAPI_DP_OPS`	`MFLOPS`
20	2	9,732	20,160,224	2,071
20	4	17,855	57,044,352	3,196
20	8	39,251	158,760,416	4,044
400	2	1,797,470	9,019,318,304	5,017
400	4	5,317,391	28,468,128,960	5,353
400	8	18,226,300	99,857,240,288	5,478
1000	2	11,298,878	56,568,118,304	5,006
1000	4	33,579,861	179,108,928,960	5,333
1000	8	117,416,843	632,125,881,248	5,383
2000	2	44,766,803	226,536,118,304	5,060
2000	4	133,577,876	718,016,928,960	5,375
2000	8	467,818,741	2,539,240,281,248	5,427

`PAPI_L1_DCM`	`PAPI_L2_DCM`	`PAPI_TOT_INS`	`PAPI_BR_MSP`	`PAPI_VEC_DP`
77,441	961	38,985,087	4,881	14,652,196
77,497	969	99,411,281	17,759	41,473,708
85,157	975	264,295,320	77,056	115,437,664
92,091,215	48,302,660	16,780,389,759	229,437	6,553,690,516
118,439,617	48,308,443	48,546,434,684	236,622	20,698,277,740
173,991,873	48,393,827	163,758,736,097	31,922,973	72,617,620,552
612,925,298	301,103,582	105,189,324,891	551,172	41,104,090,516
875,481,753	301,349,908	305,322,423,592	549,376	130,224,677,740
1,453,542,223	302,653,984	1,036,365,366,941	180,126,447	459,691,541,392

2,288,578,088	1,205,645,558	421,177,418,641	1,197,544	164,608,090,516
2,946,275,743	1,224,808,596	1,223,842,345,290	1,286,755	522,048,677,740
4,535,813,461	1,968,138,820	4,162,709,767,369	670,605,915	1,846,574,741,392

Table 3: Full performance data of `kernel_opt` after computational optimisation

3. Vectorisation with AVX2 and OpenMP

Motivation: compare the performance difference between manual and compiler vectorisation.

3.1. Manual Vectorisation

To manually vectorise the code, the AVX2 intrinsic functions are used to vectorise the code. The manually vectorised code can be found in `code_cloth_sse.cpp` To compile the code, we use compiler instruction `-march=core-avx2 -O3` to enable AVX2 intrinsic functions support.

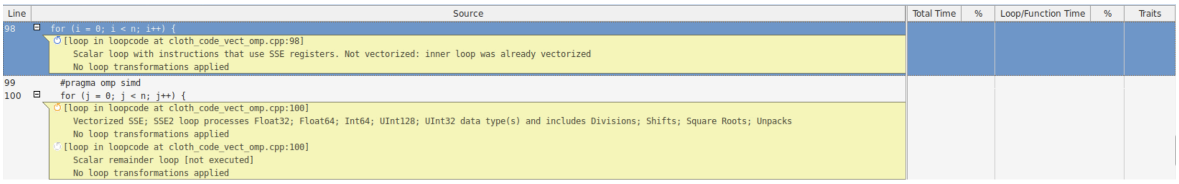
To optimise the code with AVX2 intrinsics, we manually vectorised most of the loops by unrolling loops to handle four items at a time. The memory alignment for all the arrays are also employed to maximise the memory efficiency.

3.2. Compiler Vectorisation

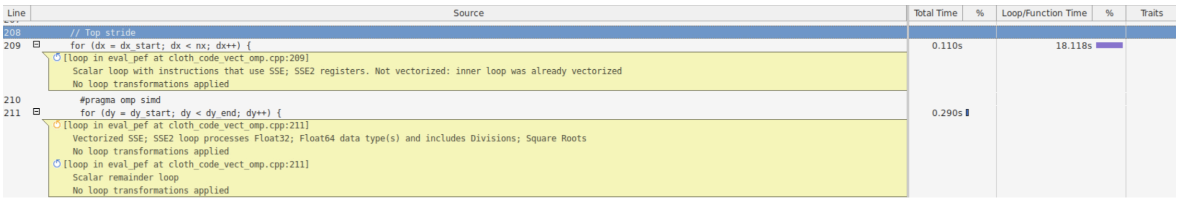
The openmp vectorised code can be found in `code_cloth_vect_omp.cpp`. `#pragma omp parallel` is used to vectorise different loops. There are seven loops intended to be vectorised with openmp's vectorisation directive, the vectorisation can be checked via intel advisor. The profile setting is: `n = 1000, d = 8, s = 1.0, m = 1.0, f = 10.0, g = 0.981, b = 3.0, o = 0.0, t = 0.05, i = 100`.

There are seven loops in the loop code (we use Intel Advisor to help to analyse the compiler-vectorisation):

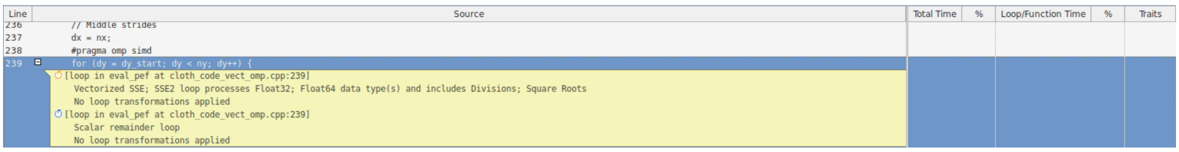
1. The loop updates position of the node and update velocity if the node collide with the ball was successfully vectorised.



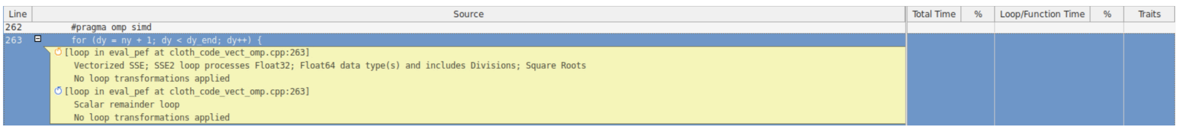
2. The loop pre-computes the reference distance was successfully vectorised.



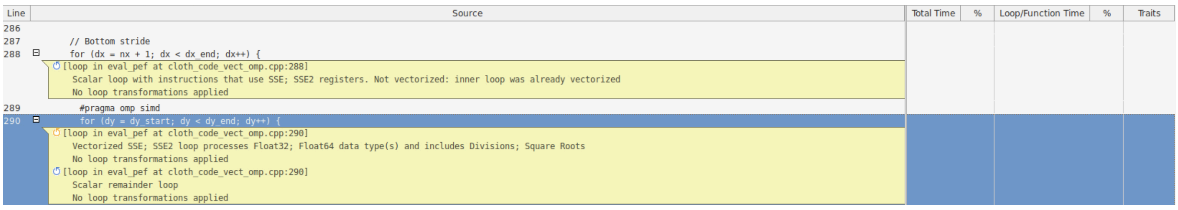
3. The loop updates energy and force from top stride was successfully vectorised.



4. The loop updates energy and force from middle-left stride was successfully vectorised.



5. The loop updates energy and force from middle-right stride was successfully vectorised.



6. The loop updates energy and force from bottom stride was not vectorised.

From the below compiler message, since there is no data dependency exist between iterations, the reason for failure vectorisation might be the loop index is too complex for compiler to analyse.

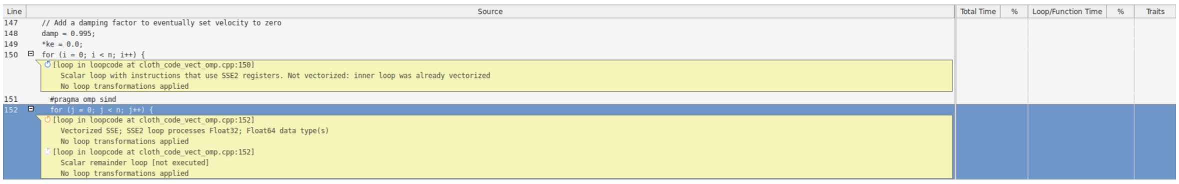
Source Code

```
int size = (2 * delta + 1);
for (dx = -delta; dx <= delta; dx++) {
  #pragma omp simd
  for (dy = -delta; dy <= delta; dy++) {
    loop_idx = (dx + delta) * size + (dy + delta);
    rlen_table[loop_idx] = sqrt((double) (dx * dx + dy * dy)) * sep;
  }
}
```

Error Message

```
cloth_code_vect_omp.cpp:151:5: warning: loop not vectorised: the optimizer was
unable to perform the requested transformation; the transformation might be
disabled or specified as part of an unsupported transformation ordering [-Wpass-
failed=transform-warning]
#pragma omp simd
```

7. The loop adds a damping factor to set velocity to zero and calculate kinetic energy was successfully vectorised.



3.3. Performance Comparison

The performance data was collected for manual vectorised (`kernel_sse`) and openmp (`kernel_vect_omp`) vectorised code and shown in Table 4, Table 5 and plotted in Figure 4. After some inspecting tables and charts, we can observe that the code uses AVX2 intrinsic functions (`kernel_sse`) provides an average of 62% speed up, especially for d is large, because when d is small the vectorisation will not apply.

Similarly, the openmp vectorised code provides an approximately the same speed up, the performance difference to `kernel_sse` is marginal (less than 1%), the 1% difference may result from not vectorising the loop for calculating reference length.

We can also observe that as the level of node interaction level increases, there is a corresponding increase in the MFLOPs. This can be attributed to the vectorisation, which yields computational benefits for the programme.

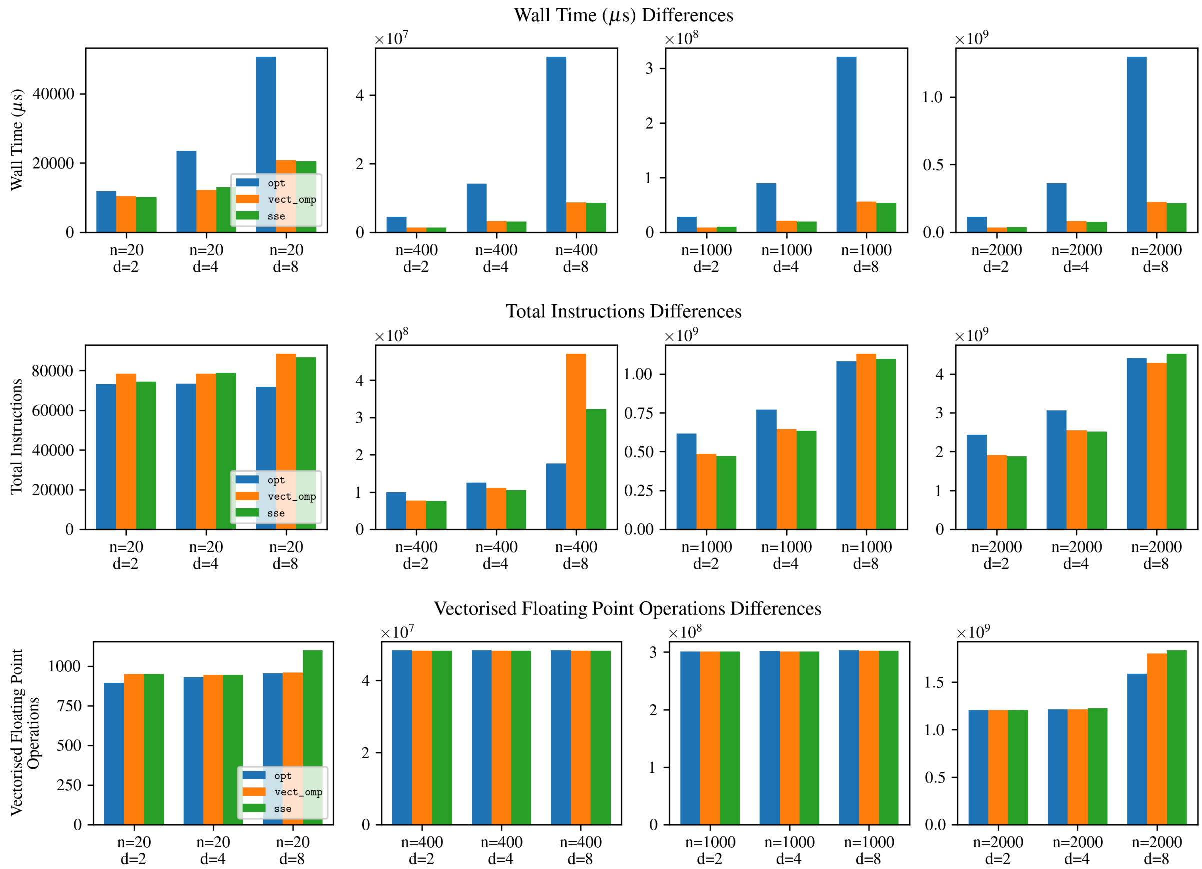


Figure 4: Performance comparison between `kernel_main` and `kernel_opt` after computational optimisation

n	d	wall time (μs)	`PAPI_DP_OPS`	`MFLOPS`
20	2	10,100	21,025,228	2,081
20	4	13,034	57,910,172	4,443
20	8	20,484	159,665,596	7,794
400	2	1,421,751	9,355,357,516	6,580
400	4	3,106,217	28,804,178,012	9,273
400	8	8,573,954	100,193,295,484	11,685
1000	2	10,086,937	58,668,157,516	5,816
1000	4	19,584,695	181,208,978,012	9,252
1000	8	53,879,769	634,225,936,060	11,771
2000	2	37,561,756	234,936,157,516	6,254
2000	4	77,556,343	726,416,978,012	9,366
2000	8	215,980,859	2,547,640,336,060	11,795

`PAPI_L1_DCM`	`PAPI_L2_DCM`	`PAPI_TOT_INS`	`PAPI_BR_MSP`	`PAPI_VEC_DP`

74,502	949	31,802,319	4,663	10,537,832
78,840	944	51,728,870	5,831	20,575,668
86,707	1,099	100,576,333	43,982	52,340,924
75,676,911	48,220,481	13,255,816,416	204,290	4,508,182,904
104,524,778	48,218,680	22,876,870,459	253,770	9,388,760,628
322,473,023	48,245,458	53,830,293,211	360,864	29,384,198,396
473,884,541	301,014,372	83,029,725,043	533,123	28,235,422,904
633,630,846	301,257,855	143,502,039,754	643,481	58,916,720,628
1,096,445,281	302,487,403	339,438,308,953	809,553	185,462,558,540
1,881,235,990	1,204,928,992	332,358,363,234	964,948	113,020,822,904
2,519,030,065	1,225,331,400	574,702,873,429	1,258,977	235,983,320,628
4,522,662,934	1,835,594,093	1,361,770,605,623	1,686,373	744,273,158,540

Table 4: Full performance data of `kernel_sse`

n	d	**wall time (μs)**	`PAPI_DP_OPS`	`MFLOPS`
20	2	10,450	22,627,428	2,165
20	4	12,261	63,707,572	5,195
20	8	20,875	172,636,596	8,270
400	2	1,340,346	10,162,369,716	7,581
400	4	3,254,501	31,640,481,412	9,722
400	8	8,758,119	107,785,596,612	12,306
1000	2	8,734,547	63,745,669,716	7,298
1000	4	20,998,577	199,039,701,412	9,478
1000	8	55,880,509	682,185,939,684	12,207
2000	2	34,650,715	255,291,169,716	7,367
2000	4	82,561,317	797,878,401,412	9,664
2000	8	224,409,001	2,740,159,839,684	12,210

`PAPI_L1_DCM`	`PAPI_L2_DCM`	`PAPI_TOT_INS`	`PAPI_BR_MSP`	`PAPI_VEC_DP`
78,475	948	28,977,252	4,495	11,688,932
78,525	945	52,918,709	5,361	23,861,268
88,551	958	102,268,065	63,714	58,771,124
76,601,552	48,201,786	12,158,485,873	274,509	5,063,070,004
110,786,192	48,212,869	23,458,015,623	288,677	10,928,994,228
471,154,950	48,229,637	53,919,596,504	404,883	32,840,658,628
486,110,147	301,005,690	76,175,114,855	659,273	31,722,630,004
644,547,106	301,208,131	147,143,805,046	729,691	68,587,314,228
1,131,102,428	302,582,051	339,811,913,646	848,815	207,243,619,396
1,911,306,132	1,204,370,817	304,949,154,826	1,565,603	126,995,230,004

2,548,394,380	1,214,954,668	589,286,555,938	1,478,609	274,724,514,228
4,285,865,351	1,800,664,186	1,363,018,807,227	1,917,327	831,635,219,396

Table 5: Full performance data of `kernel_vect_omp`

4. Parallelisation using OpenMP

Motivation: analyse the performance increase after combining parallelisation and vectorisation.

To apply the parallelisation, the vectorisation code was modified accordingly. Since openmp is not supporting reduction for custom operators (AVX add intrinsic functions), we change the code accordingly. After changed the code, even though a some overhead was introduced on executing `HSUM_PD`, we avoid the race condition of `pe_vec` variable and maximises the parallel performance (no need to use lock).

before

```
DOUBLE_VEC pe_vec = SET_PD(0.0);
for (ny = 0; ny < n; ny++) {
  for (nx = 0; nx < n; nx++) {
    ...
    // Top stride
    for (dx = dx_start; dx < nx; dx++) {
      simd_bound = dy_end - ((dy_end - dy_start) % 4);
      for (dy = dy_start; dy < simd_bound; dy += 4) {
        calc_interaction_avx(dx, dy, nx, ny, n, neighbour_size, delta, fcon, x, y,
z, x_tmp_vec, y_tmp_vec, z_tmp_vec, &fx_tmp_vec, &fy_tmp_vec, &fz_tmp_vec, &pe_vec,
rlen_table);0
      }
      ...
    }
    ...
  }
}
return 0.5 * (pe + HSUM_PD(pe_vec));
```

after

```
#pragma omp parallel for ...
for (ny = 0; ny < n; ny++) {
  for (nx = 0; nx < n; nx++) {
    DOUBLE_VEC pe_vec = SET_PD(0.0);
    ...
    // Top stride
    for (dx = dx_start; dx < nx; dx++) {
      simd_bound = dy_end - ((dy_end - dy_start) % 4);
      for (dy = dy_start; dy < simd_bound; dy += 4) {
        calc_interaction_avx(dx, dy, nx, ny, n, neighbour_size, delta, fcon, x, y,
z, x_tmp_vec, y_tmp_vec, z_tmp_vec, &fx_tmp_vec, &fy_tmp_vec, &fz_tmp_vec, &pe_vec,
rlen_table);0
      }
      ...
    }
    ...
  }
  pe += HSUM_PD(pe_vec);
}
return 0.5 * pe
```

The parallelisation directive is placed in the following loops:
1. The loop updates position of the node and update velocity if the node collide.
2. The loop pre-computes the reference distance.
3. The loop updates energy and force for all the particles.
4. The loop adds a damping factor to set velocity to zero and calculates kinetic energy.

4.1. Experiment settings

Since in multi-thread environment, the openmp's result may be inaccurate, for this section, we focus on program wall time relating to different number of threads, problem size and chunk size. For this performance testing, we are using 24 cpus, 96 GB of memory.
In this section, we measured the performance data for
1. different problem size ($n = [20, 400, 1000, 2000]$, $d = [2, 4, 8]$)
   - different n and d represents different problem size
   - for n: 20: small problem, 400: medium problem, 1000: large problem, 2000: super large problem
   - for d: 2: small problem, 4: medium problem, 8: large problem
2. different number of threads ($p = [1, 4, 8, 24, 48]$)
   - since we will be test on a 24 cores environment, we will test the performance of the program on a multiple of 24
   - for different p, we have below test purposes:
     - 1: test the performance for different scheduling strategy
     - 4 & 8: test the performance of correlation between different problem size
     - 24: mainly test the performance if we use all the available threads and the thread overhead
     - 24: mainly test the performance if we use extra number of threads thread overhead
3. different scheduling strategies (dynamic, static with chunk size = $\left[1, 4, 16, 64, \frac{n}{p}\right]$)
   - the purpose of different scheduling strategies are listed below.
     - `dynamic`: let openmp to decide scheduling
     - `static, [1|4]`: small chunk size
     - `static, [16|64]`: round robin scheduling with medium and large chunk size
     - `static, n/p`: evenly distribute the tasks to each threads

After testing with above designed test parameter, we get below result and we can conclude following findings.

4.2. Experiment Results

4.2.1. Relation to Amdahl's Law

Amdahl's Law describes the speedup in performance that can be gained from improving a particular part of a system. It's often used to predict the maximum speedup from parallelising a program. The formula is:

$$\text{Speedup} = \frac{1}{(1 + P) + \frac{P}{S}},$$

where
- P is the proportion of the program that can be parallelised.
- S is the speedup of the parallelised part.

4.2.2. Analysis of threads utilisation

Our performance data reveals a correlation between the number of threads and program performance, as visualised in Figure 5. Several factors contribute to this performance variation:

- Concurrency: Leveraging multi-core CPUs allows for simultaneous task execution, optimising computational resources.
- Load Balancing: Distributing computational tasks across multiple threads mitigates the risk of bottlenecking.
- Resource Utilisation: Effective allocation of CPU and memory resources minimises idle times.

Incorporating Amdahl's Law, these elements contribute to the P variable, which represents the proportion of the program that benefits from parallelisation. This allows us to theoretically estimate the upper limit of achievable speedup.

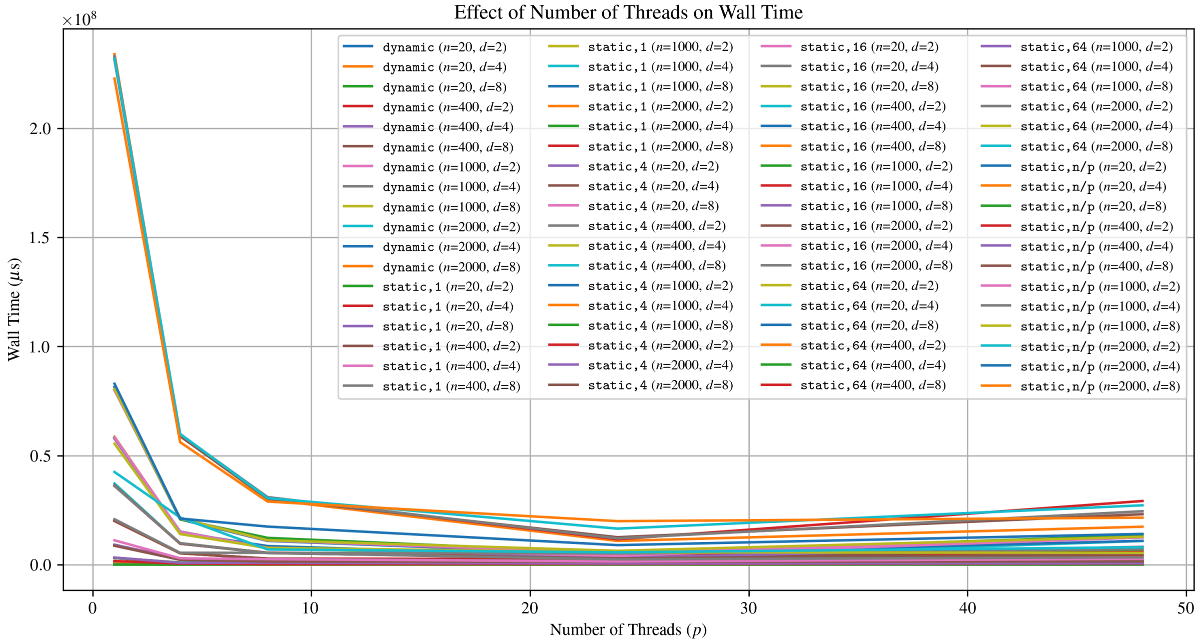


Figure 5: Performance comparison between `kernel_main` and `kernel_opt` after computational optimisation

However, an excess of threads introduces new performance issue:

- Thread Creation and Termination Overhead: As evidenced by the performance drop when $n = 20, d = 2$ and p transitions from 4 to 48 (Figure 6, left graph), the overhead of thread management depresses the S variable in Amdahl's Law, impeding performance gains.
- Context Switching and Resource Contention: When $n = 2000, d = 8$, a performance decline is observed between 24 and 48 threads (fig:init_threads, right graph). This aligns with Amdahl's Law by further diminishing the S value, corroborating the limitations in attainable speedup.

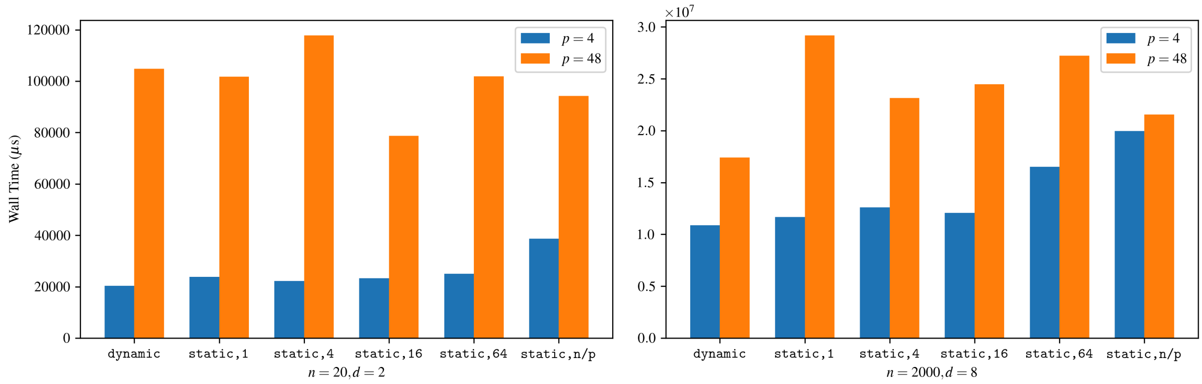


Figure 6: Performance comparison between different scheduling strategy and different number of threads

4.2.3. Analysis of Scheduling Strategies

Our findings, substantiated by Figure 7, can be summarised thus:

- Dynamic Scheduling:
  Generally, this strategy yields robust performance across an array of problem sizes and thread counts. However, the intrinsic overhead of dynamic task allocation impacts the S variable in Amdahl's Law, elucidating the performance dip as the thread count escalates.
- Static Scheduling with Fixed Chunk Sizes:
  For smaller chunk sizes (1 and 4) yields the best performance. In contrast, larger chunk sizes (16 and 64) engender workload imbalances, thereby reducing the P variable in Amdahl's Law and limiting speedup potential.
- Static Scheduling with Dynamic Chunk Size (`static,n/p`):
  This strategy yields suboptimal results. The inefficiency in chunk size allocation leads to unbalanced workload (especially in the loop for updating velocity), which affects both P and S in Amdahl's Law.

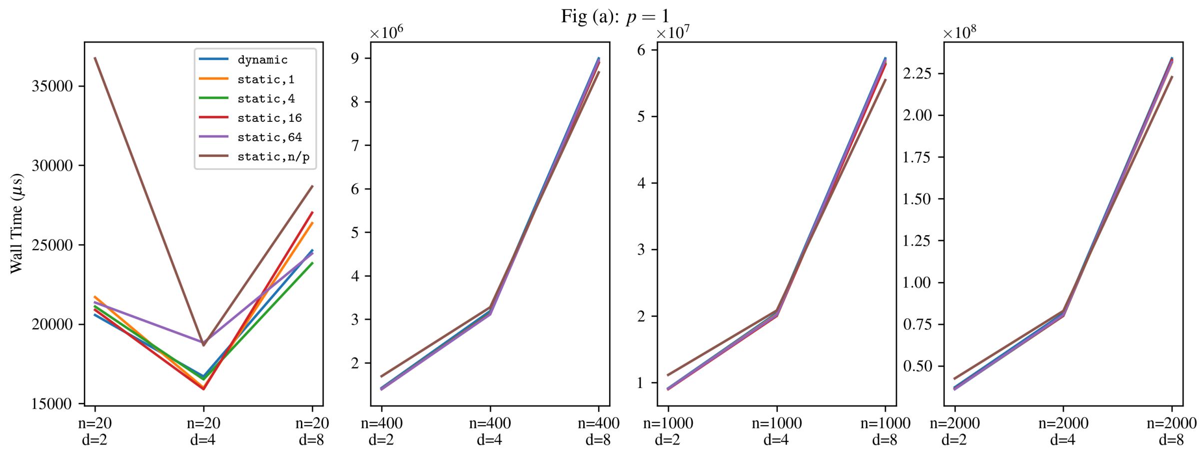


Fig (b): $p = 4$

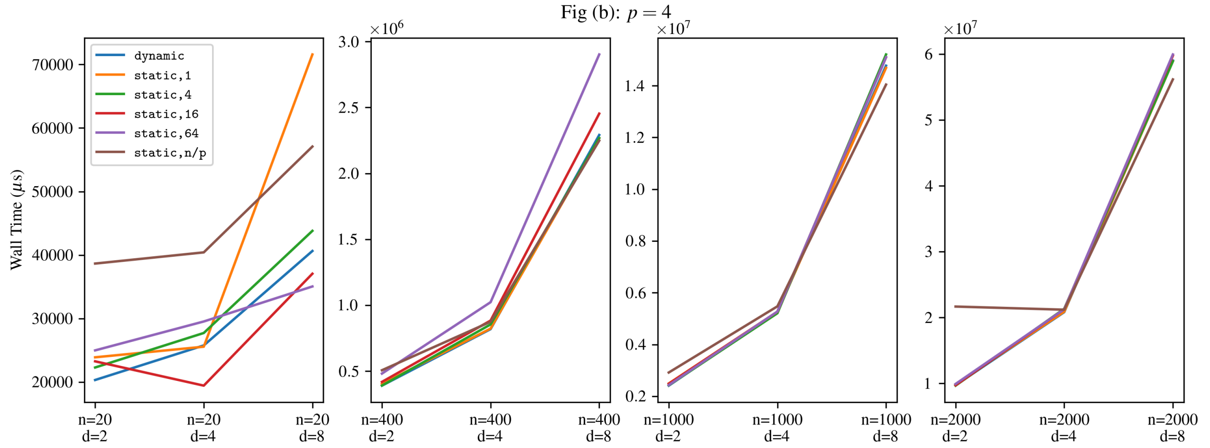


Fig (c): $p = 8$

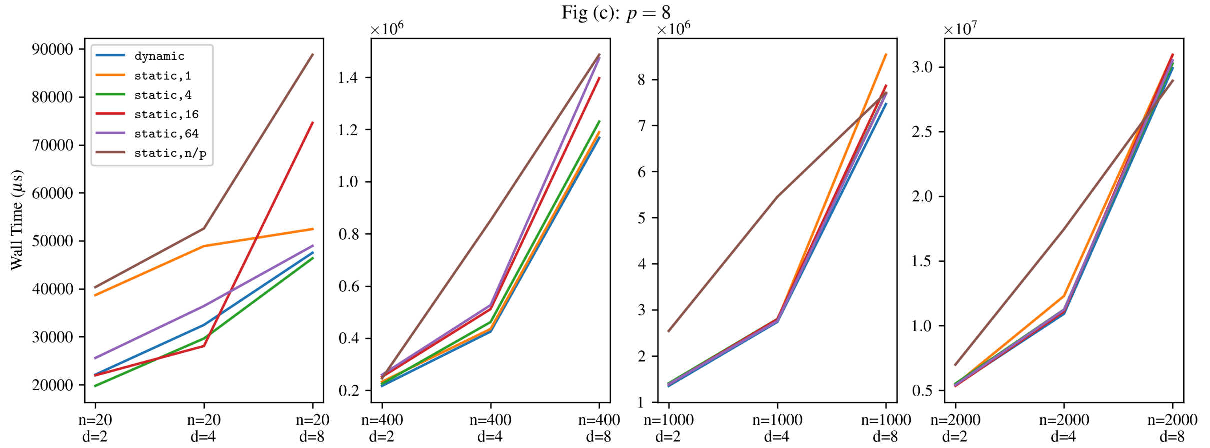


Fig (d): $p = 24$

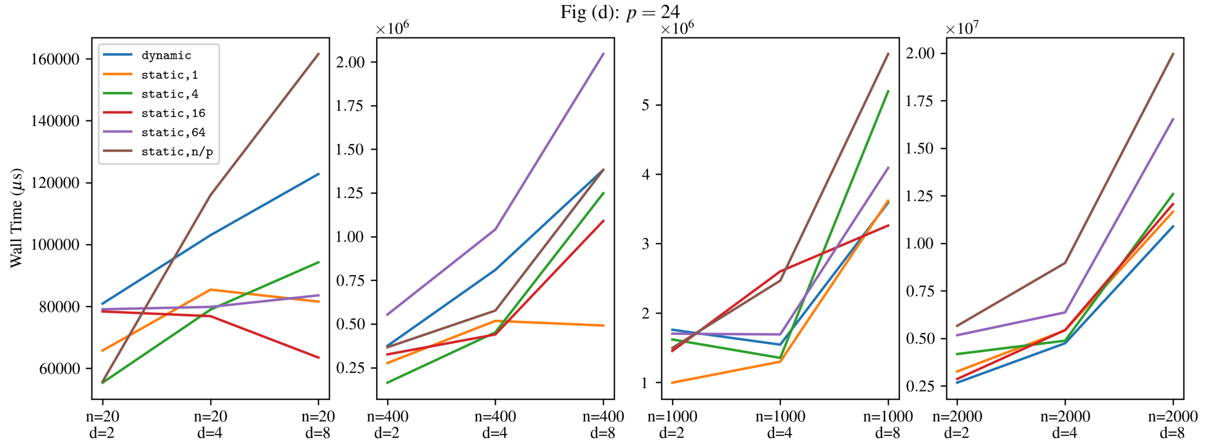


Fig (e): $p = 48$

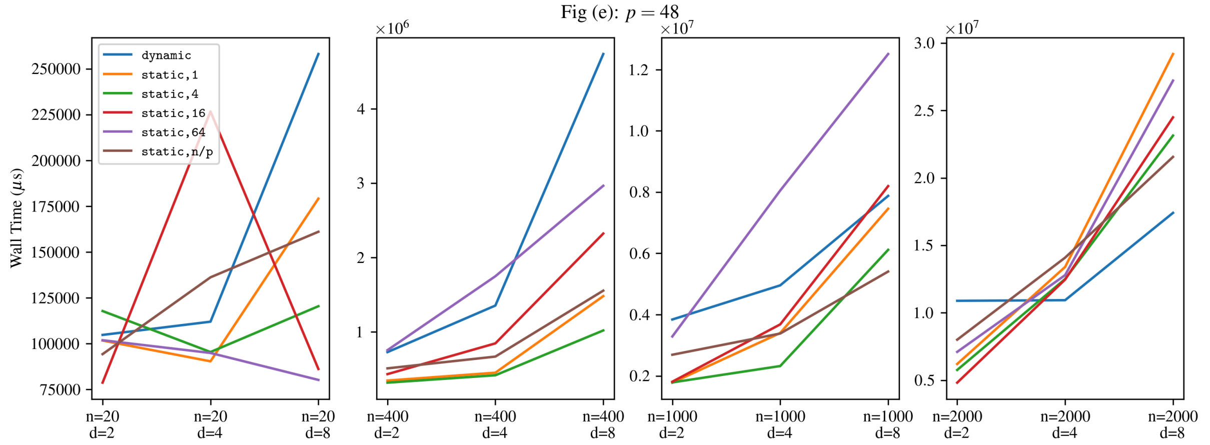


Figure 7: Performance comparison between different scheduling strategy, number of threads and problem size

n	d	p	wall time (μs)
20	2	1	20,575
20	2	4	20,313
20	2	8	22,108
20	2	24	80,917
20	2	48	104,758
20	4	1	16,706
20	4	4	25,756
20	4	8	32,468
20	4	24	103,020
20	4	48	111,990
20	8	1	24,637
20	8	4	40,633
20	8	8	47,511
20	8	24	122,779
20	8	48	258,142
400	2	1	1,422,748
400	2	4	390,445
400	2	8	217,211
400	2	24	374,667
400	2	48	728,780
400	4	1	3,188,775
400	4	4	818,999
400	4	8	425,384
400	4	24	809,585
400	4	48	1,354,477
400	8	1	8,995,582
400	8	4	2,291,258
400	8	8	1,168,005
400	8	24	1,382,027
400	8	48	4,734,391
1000	2	1	9,153,501
1000	2	4	2,411,892
1000	2	8	1,348,101
1000	2	24	1,759,050
1000	2	48	3,841,835
1000	4	1	20,449,825

n	d	p	wall time (μs)
20	2	1	21,705
20	2	4	23,886
20	2	8	38,677
20	2	24	65,766
20	2	48	101,690
20	4	1	15,991
20	4	4	25,557
20	4	8	48,900
20	4	24	85,432
20	4	48	90,344
20	8	1	26,359
20	8	4	71,579
20	8	8	52,453
20	8	24	81,564
20	8	48	179,187
400	2	1	1,400,856
400	2	4	403,628
400	2	8	232,894
400	2	24	276,295
400	2	48	346,863
400	4	1	3,133,256
400	4	4	823,238
400	4	8	435,493
400	4	24	517,547
400	4	48	452,382
400	8	1	8,910,027
400	8	4	2,266,167
400	8	8	1,189,621
400	8	24	491,245
400	8	48	1,481,107
1000	2	1	9,076,584
1000	2	4	2,494,697
1000	2	8	1,387,939
1000	2	24	996,973
1000	2	48	1,780,709
1000	4	1	20,261,717

1000	4	4	5,258,535
1000	4	8	2,736,289
1000	4	24	1,544,504
1000	4	48	4,956,918
1000	8	1	58,682,230
1000	8	4	14,767,413
1000	8	8	7,465,662
1000	8	24	3,589,072
1000	8	48	7,876,233
2000	2	1	37,189,451
2000	2	4	9,688,201
2000	2	8	5,387,113
2000	2	24	2,673,703
2000	2	48	10,889,425
2000	4	1	81,395,464
2000	4	4	20,782,826
2000	4	8	10,914,305
2000	4	24	4,744,830
2000	4	48	10,934,597
2000	8	1	233,937,042
2000	8	4	59,006,004
2000	8	8	29,908,932
2000	8	24	10,890,907
2000	8	48	17,407,677

Table 6: Full performance data of `kernel_omp` with dynamic scheduling

n	d	p	wall time (μs)
20	2	1	21,113
20	2	4	22,280
20	2	8	19,770
20	2	24	55,382
20	2	48	117,803
20	4	1	16,522
20	4	4	27,716
20	4	8	29,643
20	4	24	78,978
20	4	48	95,424
20	8	1	23,831

1000	4	4	5,262,790
1000	4	8	2,768,480
1000	4	24	1,298,785
1000	4	48	3,402,516
1000	8	1	58,099,455
1000	8	4	14,681,499
1000	8	8	8,536,025
1000	8	24	3,617,843
1000	8	48	7,456,924
2000	2	1	36,246,517
2000	2	4	9,808,754
2000	2	8	5,428,900
2000	2	24	3,264,582
2000	2	48	6,207,433
2000	4	1	80,119,978
2000	4	4	20,854,305
2000	4	8	12,291,015
2000	4	24	5,425,747
2000	4	48	13,397,818
2000	8	1	231,330,593
2000	8	4	58,896,191
2000	8	8	30,932,546
2000	8	24	11,666,279
2000	8	48	29,181,762

Table 7: Full performance data of `kernel_omp` with static scheduling, chunk size 1

n	d	p	wall time (μs)
20	2	1	20,900
20	2	4	23,254
20	2	8	21,962
20	2	24	78,378
20	2	48	78,687
20	4	1	15,904
20	4	4	19,426
20	4	8	28,073
20	4	24	76,871
20	4	48	226,745
20	8	1	27,016

20	8	4	43,803
20	8	8	46,366
20	8	24	94,266
20	8	48	120,434
400	2	1	1,395,804
400	2	4	391,040
400	2	8	225,254
400	2	24	163,928
400	2	48	318,371
400	4	1	3,136,697
400	4	4	854,478
400	4	8	461,871
400	4	24	452,017
400	4	48	417,808
400	8	1	8,886,268
400	8	4	2,268,288
400	8	8	1,230,111
400	8	24	1,248,296
400	8	48	1,020,054
1000	2	1	9,121,846
1000	2	4	2,424,743
1000	2	8	1,404,567
1000	2	24	1,619,137
1000	2	48	1,789,023
1000	4	1	20,022,662
1000	4	4	5,207,751
1000	4	8	2,798,884
1000	4	24	1,353,660
1000	4	48	2,323,638
1000	8	1	57,864,472
1000	8	4	15,199,740
1000	8	8	7,692,072
1000	8	24	5,192,935
1000	8	48	6,115,861
2000	2	1	36,133,792
2000	2	4	9,620,545
2000	2	8	5,514,223
2000	2	24	4,178,248
2000	2	48	5,764,022

20	8	4	37,054
20	8	8	74,597
20	8	24	63,489
20	8	48	86,143
400	2	1	1,393,901
400	2	4	418,744
400	2	8	251,626
400	2	24	325,805
400	2	48	432,933
400	4	1	3,123,750
400	4	4	883,867
400	4	8	511,634
400	4	24	439,656
400	4	48	846,311
400	8	1	8,898,711
400	8	4	2,453,220
400	8	8	1,396,532
400	8	24	1,089,722
400	8	48	2,321,994
1000	2	1	9,019,100
1000	2	4	2,485,728
1000	2	8	1,379,691
1000	2	24	1,453,929
1000	2	48	1,812,275
1000	4	1	20,025,225
1000	4	4	5,266,946
1000	4	8	2,795,505
1000	4	24	2,599,503
1000	4	48	3,683,358
1000	8	1	57,748,549
1000	8	4	15,089,625
1000	8	8	7,860,866
1000	8	24	3,259,743
1000	8	48	8,195,575
2000	2	1	36,335,182
2000	2	4	9,677,020
2000	2	8	5,340,840
2000	2	24	2,867,517
2000	2	48	4,821,824

2000	4	1	79,923,767
2000	4	4	21,297,487
2000	4	8	11,244,257
2000	4	24	4,880,984
2000	4	48	12,506,188
2000	8	1	232,701,511
2000	8	4	58,878,482
2000	8	8	30,271,060
2000	8	24	12,595,036
2000	8	48	23,137,069

Table 8: Full performance data of `kernel_omp` with static scheduling, chunk size 4

n	d	p	wall time (μs)
20	2	1	21,365
20	2	4	24,976
20	2	8	25,584
20	2	24	79,099
20	2	48	101,869
20	4	1	18,836
20	4	4	29,527
20	4	8	36,408
20	4	24	79,854
20	4	48	94,899
20	8	1	24,451
20	8	4	35,044
20	8	8	48,952
20	8	24	83,581
20	8	48	80,184
400	2	1	1,396,148
400	2	4	481,769
400	2	8	259,620
400	2	24	553,831
400	2	48	755,162
400	4	1	3,112,477
400	4	4	1,023,209
400	4	8	526,741
400	4	24	1,041,229
400	4	48	1,748,044

2000	4	1	80,037,834
2000	4	4	21,210,978
2000	4	8	11,089,772
2000	4	24	5,440,434
2000	4	48	12,461,923
2000	8	1	232,925,621
2000	8	4	59,725,592
2000	8	8	30,945,510
2000	8	24	12,067,770
2000	8	48	24,487,522

Table 9: Full performance data of `kernel_omp` with static scheduling, chunk size 16

n	d	p	wall time (μs)
20	2	1	36,732
20	2	4	38,650
20	2	8	40,343
20	2	24	55,731
20	2	48	94,262
20	4	1	18,661
20	4	4	40,412
20	4	8	52,571
20	4	24	115,960
20	4	48	136,209
20	8	1	28,665
20	8	4	57,078
20	8	8	88,794
20	8	24	161,557
20	8	48	161,092
400	2	1	1,691,954
400	2	4	507,266
400	2	8	246,359
400	2	24	366,521
400	2	48	510,706
400	4	1	3,276,416
400	4	4	874,692
400	4	8	851,851
400	4	24	576,547
400	4	48	668,826

400	8	1	8,952,688
400	8	4	2,901,943
400	8	8	1,473,116
400	8	24	2,045,281
400	8	48	2,964,003
1000	2	1	9,098,493
1000	2	4	2,434,088
1000	2	8	1,385,062
1000	2	24	1,702,416
1000	2	48	3,287,712
1000	4	1	20,161,500
1000	4	4	5,272,873
1000	4	8	2,756,192
1000	4	24	1,691,771
1000	4	48	8,051,762
1000	8	1	58,392,933
1000	8	4	15,092,255
1000	8	8	7,680,203
1000	8	24	4,089,141
1000	8	48	12,505,930
2000	2	1	36,070,369
2000	2	4	9,875,144
2000	2	8	5,406,673
2000	2	24	5,169,634
2000	2	48	7,102,625
2000	4	1	80,476,374
2000	4	4	21,215,533
2000	4	8	11,225,547
2000	4	24	6,369,107
2000	4	48	12,791,626
2000	8	1	231,862,282
2000	8	4	59,919,424
2000	8	8	30,505,815
2000	8	24	16,525,686
2000	8	48	27,209,710

Table 10: Full performance data of `kernel_omp` with static scheduling, chunk size 64

400	8	1	8,675,149
400	8	4	2,248,521
400	8	8	1,486,411
400	8	24	1,382,108
400	8	48	1,554,869
1000	2	1	11,164,091
1000	2	4	2,917,978
1000	2	8	2,539,787
1000	2	24	1,496,689
1000	2	48	2,691,752
1000	4	1	20,833,983
1000	4	4	5,473,749
1000	4	8	5,451,051
1000	4	24	2,466,650
1000	4	48	3,384,611
1000	8	1	55,426,798
1000	8	4	14,037,047
1000	8	8	7,709,632
1000	8	24	5,730,157
1000	8	48	5,406,891
2000	2	1	42,481,581
2000	2	4	21,629,612
2000	2	8	6,979,923
2000	2	24	5,663,485
2000	2	48	8,006,147
2000	4	1	82,894,973
2000	4	4	21,153,815
2000	4	8	17,458,500
2000	4	24	8,968,028
2000	4	48	14,089,273
2000	8	1	222,734,974
2000	8	4	56,133,271
2000	8	8	28,929,986
2000	8	24	19,956,559
2000	8	48	21,556,926

Table 11: Full performance data of `kernel_omp` with static scheduling, chunk size $\frac{n}{p}$

5. Roofline analysis

Motivation: find the performance bottleneck of the program (whether it's compute-bounded or IO-bounded).

The intel advisor is used to generate the roofline plot shown in roofline. The profile settings is as follows: $n = 2000$ $d = 8$ $p = 48$ $s = 1.0$, $m = 1.0$, $f = 10.0$, $g = 0.981$, $b = 3.0$, $o = 0.0$, $t = 0.05$, and $i = 100$.

From the plot, we can see that the yellow and red dots (corresponds to `calc_interaction_avx` function) are closer to the peak computational performance line (the upper horizontal line), hence `calc_interaction_avx` is computational bounded. In addition, since the function `calc_interaction_avx` is nested in the most computationally intensive ($\mathcal{O}(n^2d^2)$ complexity) loop, hence we conclude that the program is compute bounded.

To even further improve the performance of the current program, we can use AVX-512 intrinsics function, as it will maximise the power of SIMD, however, it may introduce other performance degrade problem caused by overheat.

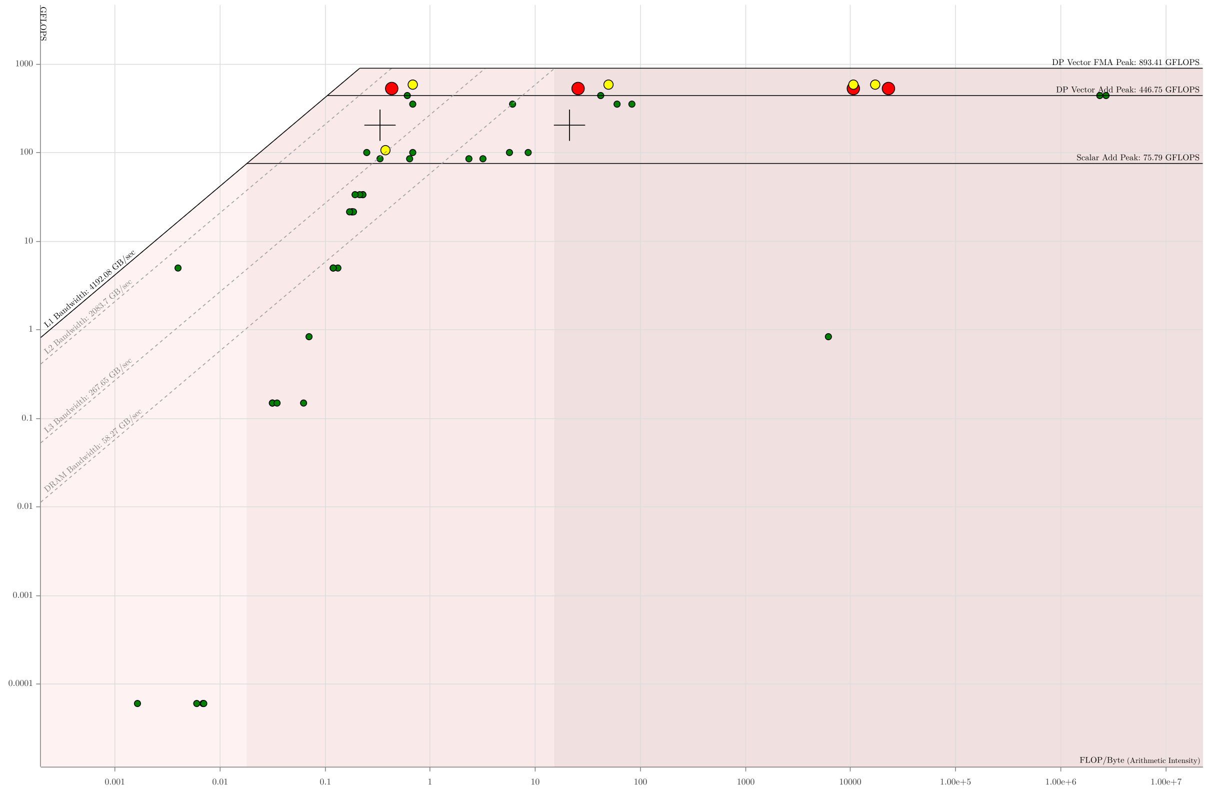


Figure 8: Roofline plot for `kernel_omp`

6. Appendix

Name	Meaning
n	Nodes per dimenstion
d	Node interaction level
p	Maximum number of OpenMP threads
wall time (μs)	Program wall time after i iterations
`PAPI_DP_OPS`	Floating point operations; optimized to count scaled double precision vector operations
`PAPI_L1_DCM`	Level 1 data cache misses
`PAPI_L2_DCM`	Level 2 data cache misses
`PAPI_TOT_INS`	Instructions completed
`PAPI_BR_MSP`	Conditional branch instructions mispredicted
`PAPI_VEC_DP`	Double precision vector/SIMD instructions

Table 12: Performance metrices names