Abstract

This work shows three optimizations for a 2D stencil computation: Parallelization with OpenMP, blocking in X and Y directions, and vectorization using SIMD processor instructions. Performance is measured on the Tsubame 2.5 supercomputer of Tokyo Institute of Technology.

1 Problem Description: Diffusion

The base problem for this assignment is an iterative diffusion (2D stencil computation). Given a 2D matrix, the result of one iteration is a new matrix of same size, but every value computed as the average of the old value and its four neighboring values. A checksum is computed to ensure correctness of the following optimized versions.

```
#define NX 8194
#define NY 8194

float data[NY][NX][2];
double checksum = 0;
int from = 0;
int to = 1;

// Initialize data[from]

for (int n = 0; n < ITER; n++) {
    for (int y = 1; y < NY - 1; y++) {
        for(x = 1; x < NX - 1; x++) {
            data[to][y][x] = 0.2 * (data[from][y][x]
                                  + data[from][y][x-1]
                                  + data[from][y][x+1]
                                  + data[from][y-1][x]
                                  + data[from][y+1][x]);
            checksum += data[to][y][x];
        }
    }

    from = (from + 1) % 2;
    to = (to + 1) % 2;
}
```

System Environment All benchmarks were run on the TSUBAME 2.5 supercomputer running Linux x86_64 with kernel version 3.0.76-0.11. The compiler for the C programs is gcc 4.3.4 and no optimization flags were used to isolate the effects of the optimzations described in this work. Benchmarks were run on a thin node with 2x Intel Xeon X5670 processors each of which has 6 cores (12 with hyperthreading), 32 KB data L1 cache per core, 256 KB L2 cache per core, 12 MB shared L3 cache, a cache line size of 64 bytes for all caches, and supports SIMD instructions through MMX and SSE (in various versions). The machine has 54 GB of main memory.

2 Parallelization with OpenMP

In this version, the outer loop of the stencil computation is parallelized with OpenMP. The program will spawn $t =$ `OMP_NUM_THREADS` many shared memory threads and every thread $i \in [0, t)$ will compute the iteration range $y = [\frac{i \cdot NY}{t}, \frac{(i+1) \cdot NY}{t})$, not accounting for rounding if NY does not divide evenly. This parallelization is done automatically by OpenMP and programmers only have to specify which loop should be parallelized and that the value `checksum` is shared by all threads and should be reduced.

This optimized version also introduces blocking in X and Y direction. Blocking divides the matrix in a 2D grid of equally-sized blocks and blocks are computed one by one. This can increase data locality and thus cache utilization. If `BLOCK_X` and `BLOCK_Y` are set to `1`, blocking is deactivated. Moreover, using OpenMP with a static scheduling strategy implicitly adds another level of blocking in Y direction.

```
#include <omp.h>

#define NX 8194
#define NY 8194
#define BLOCK_X 1
#define BLOCK_Y 1
```

```
7
8  float data[NY][NX][2];
9  double checksum = 0;
10 int from = 0;
11 int to = 1;
12
13 // Initialize data[from]
14
15 for (int n = 0; n < ITER; n++) {
16 #pragma omp parallel
17     {
18       int x, y, x_base, y_base;
19
20 #pragma omp for reduction(+:checksum)
21       for (y_base = 1; y_base < NY - 1; y_base += BLOCK_Y) {
22           for (x_base = 1; x_base < NX - 1; x_base += BLOCK_X) {
23               for(y = y_base; y < y_base + BLOCK_Y; y++) {
24                   for(x = x_base; x < x_base + BLOCK_X; x++) {
25                       data[to][y][x] = 0.2 * (data[from][y][x]
26                                             + data[from][y][x-1]
27                                             + data[from][y][x+1]
28                                             + data[from][y-1][x]
29                                             + data[from][y+1][x]);
30                       checksum += data[to][y][x];
31                   }
32               }
33           }
34       }
35     }
36
37     from = (from + 1) % 2;
38     to = (to + 1) % 2;
39 }
```

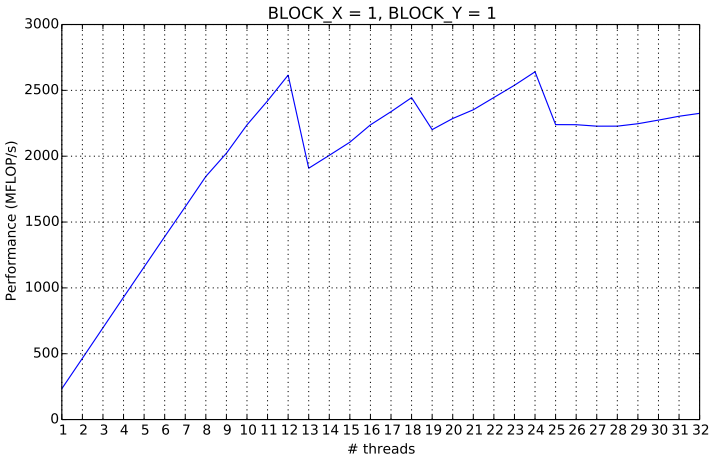


Figure 1: Performance Overview without any Blocking.

Benchmarks Tsubame 2.5 was used for benchmarking the implementation shown above. Every blocking combination (all powers of 2 up to 512 for both X and Y) was run for 1 to 32 threads on a 8194 × 8194 square matrix. The matrix size was chosen such that blocking factors always divide the matrix evenly and no special cases for the last/border blocks are required. Figure 1 gives an overview of the performance when no blocking is used (`BLOCK_X` = `BLOCK_Y` = 1). In this case, it is best to use 12 threads. This is because the a Tsubame thin node has 12 cores. Using more threads cannot increase the performance, i.e., this implementation does not benefit from hyperthreading.

Figure 2 shows an overview of the performance with only one thread but all blocking values in either X or Y direction. In this case, choosing a blocking value of `BLOCK_Y` = 2 or `BLOCK_X` = 512 is best. If blocking is allowed in both X and Y direction, `BLOCK_X` = 512 and `BLOCK_Y` = 16 reaches a performance of 283.3 MFLOP/s, which is marginally better than just blocking in a single direction. In that combination, each block has 512 × 16 elements and thus a size of 32 KB, which is the size of the L1 cache. Overall, this implementation does not benefit much from blocking when only one thread is used.

Figure 3 shows an overview of the performance with 12 threads and all blocking values in either X or Y direction. In that case, choosing a blocking value of `BLOCK_Y` = 2 or `BLOCK_X` = 512 is best. If blocking is allowed in both X and Y direction, `BLOCK_X` = 512 and `BLOCK_Y` = 1 is best at a performance of 3151.2 MFLOP/s, which is a good speedup. Using OpenMP and explicit blocking effectively results in two levels of blocking: First, every thread is assigned a block of size 8194 × (8194/12). Second, within a thread, explicit blocking is applied. It is unclear why the best performance is reached at the reported combination, because it matches neither the L1 nor the L2 cache size.

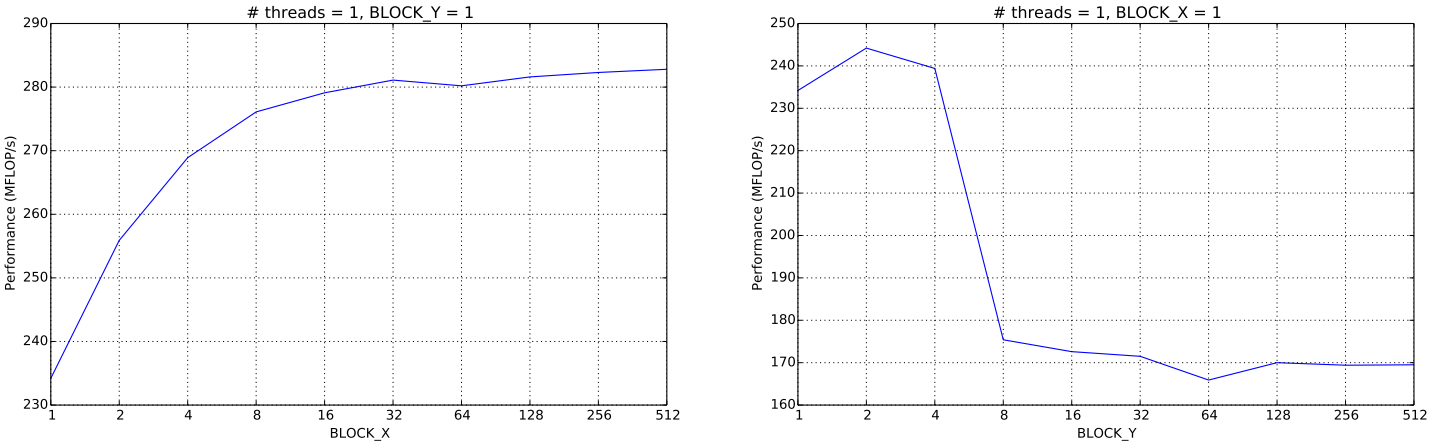


Figure 2: Performance for #threads = 1, blocking in only X or Y direction.

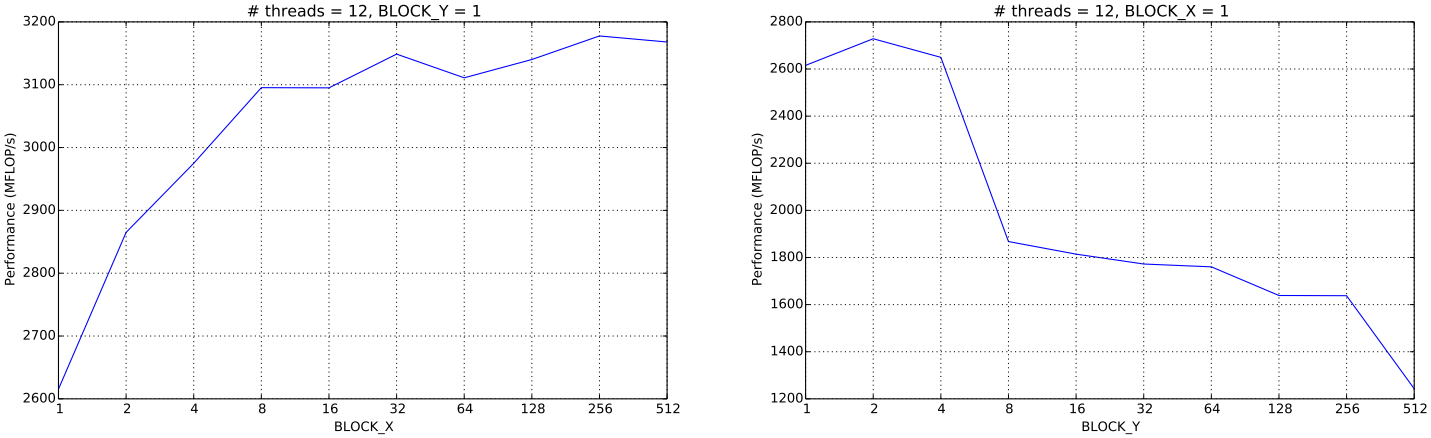


Figure 3: Performance for #threads = 12, blocking in only X or Y direction.

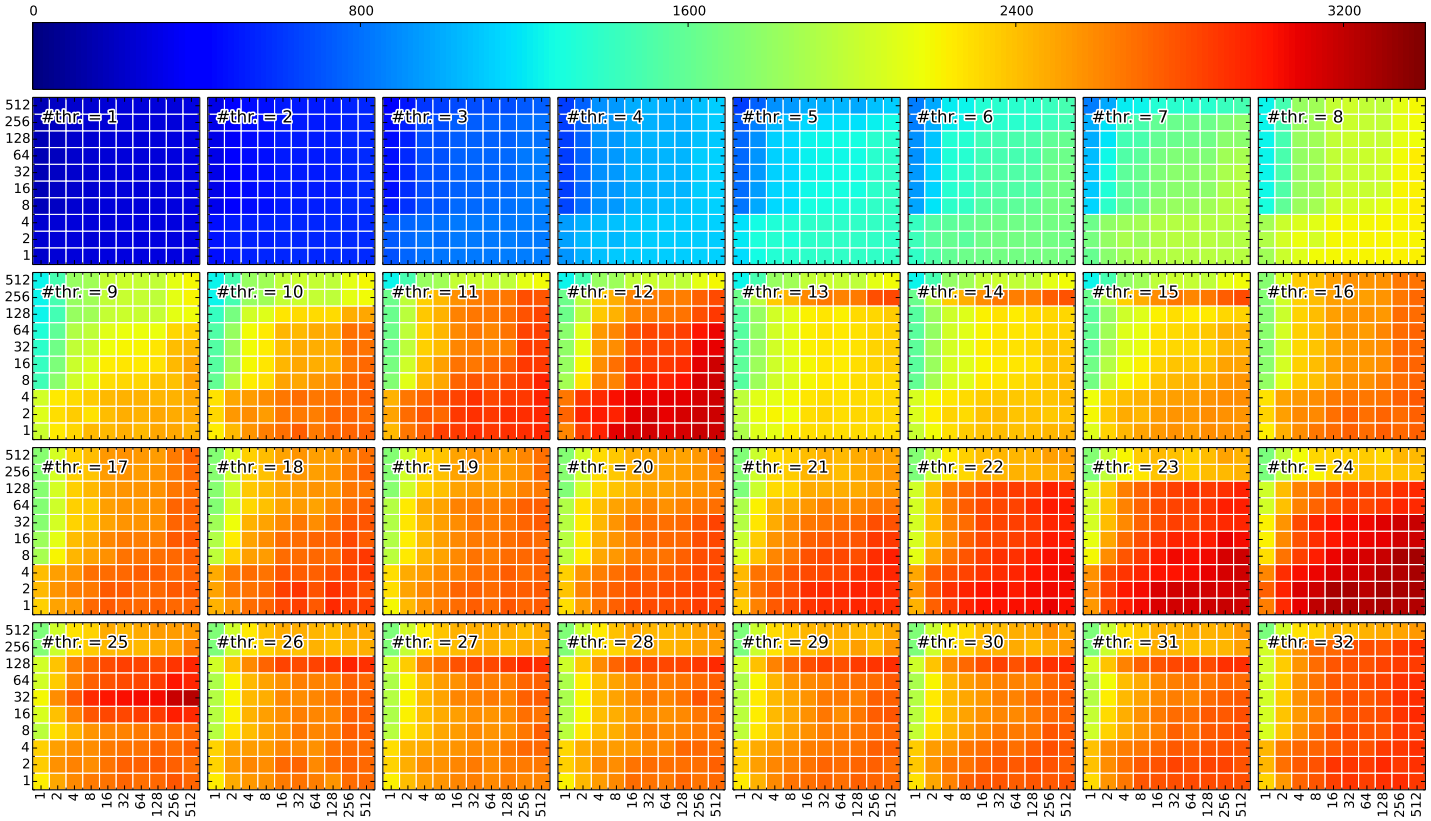


Figure 4: Performance Overview: All blocking combinations (powers of 2) in X and Y direction.

3 Loop Unrolling

In this version, the innermost loop is unrolled twice, i.e., every iteration computes two elements. This is beneficial if a processor has multiple arithmetic units per core and gives rise to instruction-level parallelism. In contrast, the speedup gained in the previous section is due to thread-level parallelism. If l is the level of unrolling of the innermost loop (here $l = 2$), then `BLOCK_X` $\geq l$. Otherwise, results will be computed redundantly.

```
#define BLOCK_X 2 // Must be >= 2
#define BLOCK_Y 1

#pragma omp for reduction(+:checksum)
for (y_base = 1; y_base < NY - 1; y_base += BLOCK_Y) {
    for (x_base = 1; x_base < NX - 1; x_base += BLOCK_X) {
        for(y = y_base; y < y_base + BLOCK_Y; y++) {
            for(x = x_base; x < x_base + BLOCK_X; x+=2) {
              data[to][y][x] = 0.2 * (data[from][y][x]
                                    + data[from][y][x-1]
                                    + data[from][y][x+1]
                                    + data[from][y-1][x]
                                    + data[from][y+1][x]);
              checksum += data[to][y][x];

              data[to][y][x+1] = 0.2 * (data[from][y][x+1]
                                    + data[from][y][x]
                                    + data[from][y][x+2]
                                    + data[from][y-1][x+1]
                                    + data[from][y+1][x+1]);
              checksum += data[to][y][x+1];
            }
        }
    }
}
```

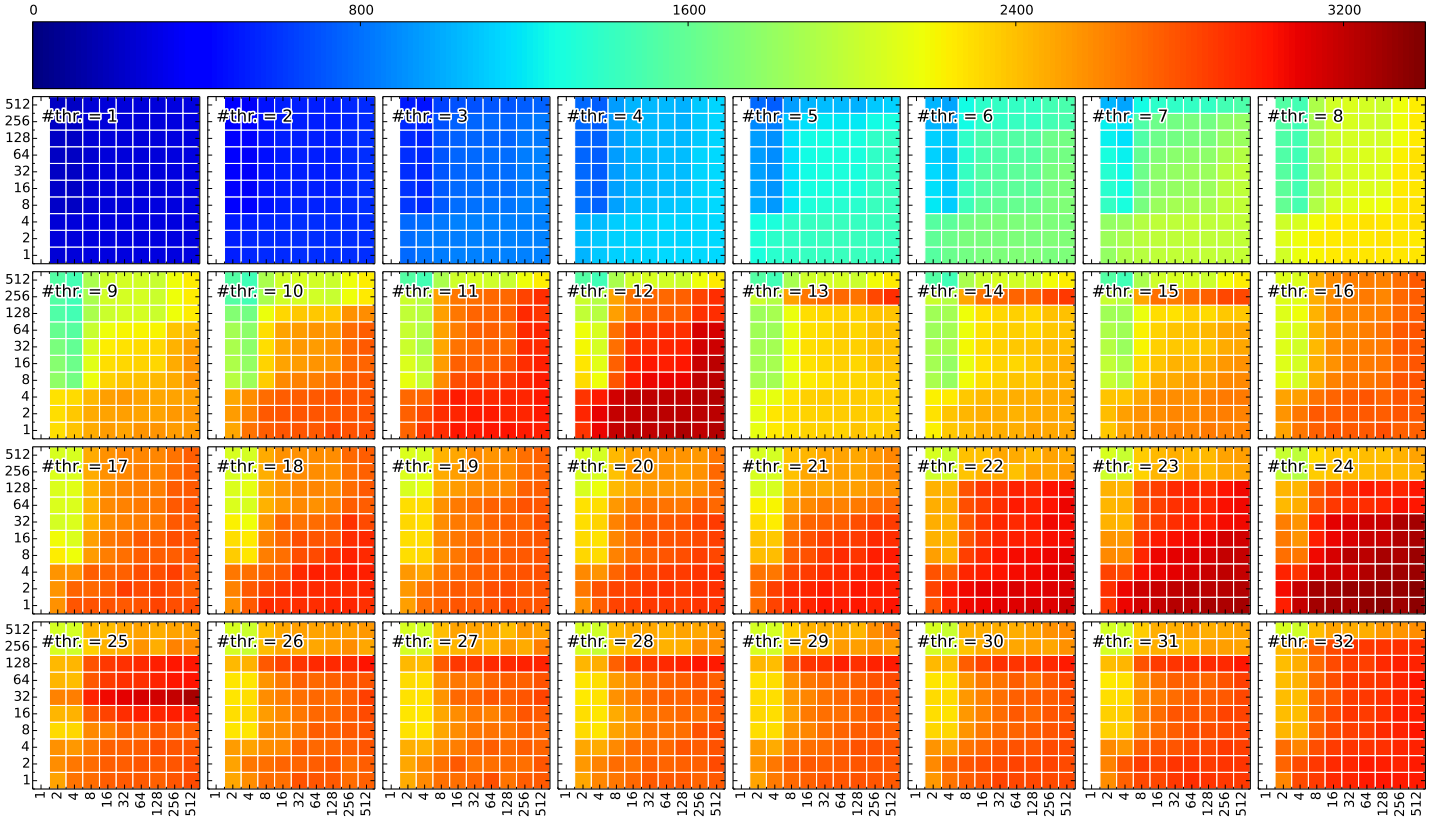
Benchmarks Unrolling iterations for X allows for thread-level parallelism but does overall not increase the performance much compared to only blocking. Figure 5 gives an overview of the performance with all combinations of blocking and either 2 or 4 unrolled iterations. The best performance with 2 unrolled iterations is reached at `BLOCK_X` $= 512$ and `BLOCK_Y` $= 2$ with 24 threads at 3368.8 MFLOP/s. The best performance with 4 unrolled iterations is reached at `BLOCK_X` $= 128$ and `BLOCK_Y` $= 1$ with 24 threads at 3396.7 MFLOP/s.

4 Vectorization of Unrolled Code

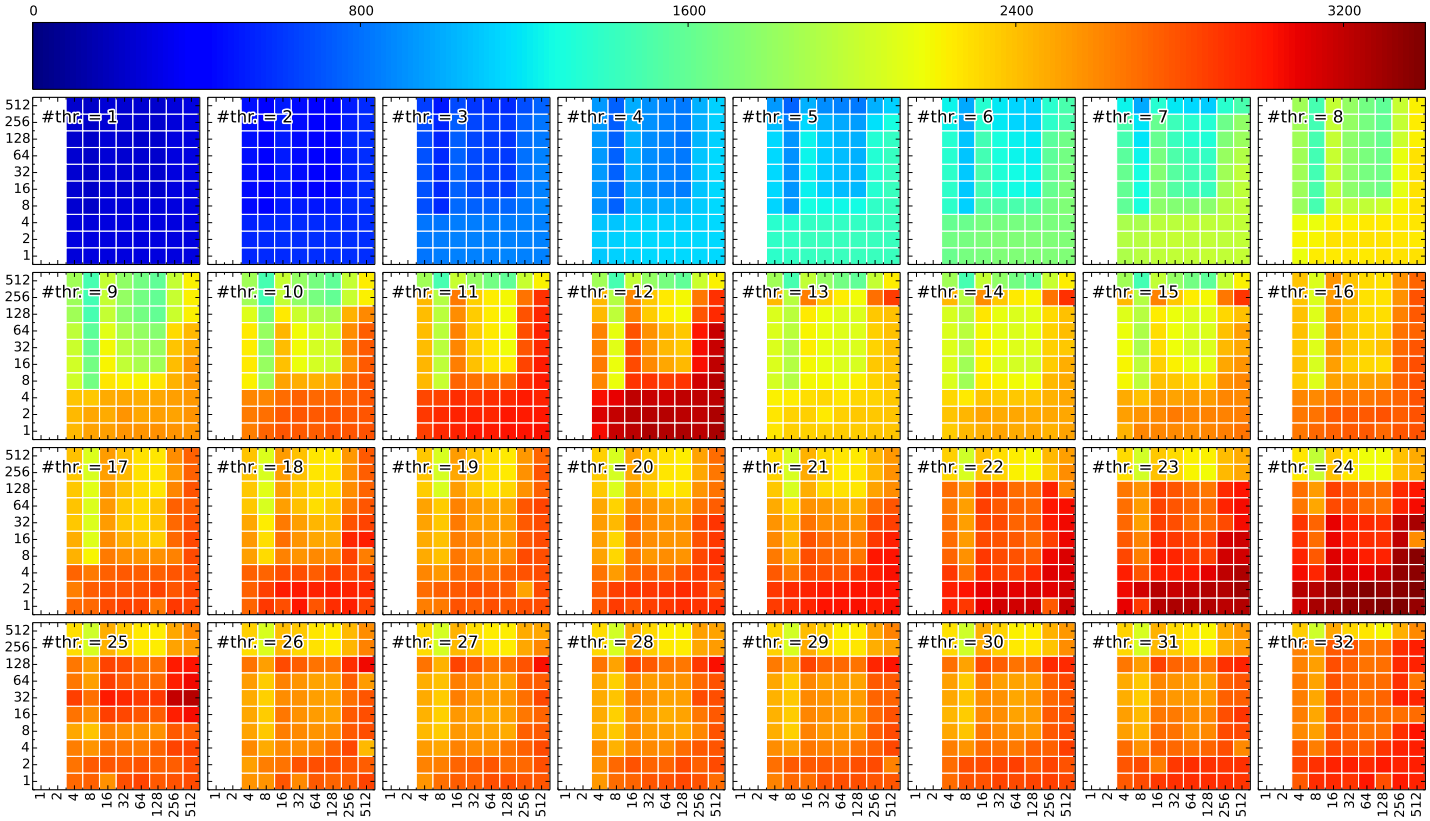
This final optimization builds on top of the previous optimization. The innermost loop is unrolled with a factor of $l = 4$ and the computation utilizes vectorized SIMD registers and instructions. The type `__m128` denotes a 128 bit value containing four 32 bit floating point numbers. After loading values into the registers[1], arithmetic operations such as `_mm_add_ps` for addition or `_mm_mul_ps` for multiplication can be performed on all four values in parallel.

```
#include <emmintrin.h>

#define BLOCK_X 4 // Must be >= 4
#define BLOCK_Y 1

const __m128 factor = _mm_set1_ps(0.2);

#pragma omp for reduction(+:checksum)
for (y_base = 1; y_base < NY - 1; y_base += BLOCK_Y) {
    for (x_base = 1; x_base < NX - 1; x_base += BLOCK_X) {
        for(y = y_base; y < y_base + BLOCK_Y; y++) {
            for(x = x_base; x < x_base + BLOCK_X; x+=4) {
                __m128 v1 = _mm_loadu_ps(&data[from][y][x]);
                __m128 v2 = _mm_loadu_ps(&data[from][y][x-1]);
                __m128 v3 = _mm_loadu_ps(&data[from][y][x+1]);
                __m128 v4 = _mm_loadu_ps(&data[from][y-1][x]);
```

[1] Operation: `_mm_loadu_ps`; the `u` is required because memory addresses are not necessarily aligned.



(a) Unrolling 2 Iterations of X



(b) Unrolling 4 Iterations of X

Figure 5: Performance Overview with Unrolling: All blocking combinations (powers of 2) in X and Y direction.

```
                __m128 v5 = _mm_loadu_ps(&data[from][y+1][x]);

                __m128 vr = _mm_mul_ps(
                    _mm_add_ps(
                        _mm_add_ps(v1, v2),
                        _mm_add_ps(v3,
                            _mm_add_ps(v4, v5))), factor);

                _mm_storeu_ps(&data[to][y][x], vr);


                checksum += data[to][y][x];
                checksum += data[to][y][x+1];
                checksum += data[to][y][x+2];
                checksum += data[to][y][x+3];
            }
        }
    }
}
```

Benchmarks Vectorization increases the performance of factor-4 unrolled code dramatically. The best performance is reached with `BLOCK_X` = 128, `BLOCK_Y` = 2 and 24 threads at 5602.2 MFLOP/s. Figure 6 gives an overview of the performance in all measured configurations. Figure 7 shows the performance overview for a smaller matrix of size 4098 × 4098. The best performance is reached with `BLOCK_X` = 256, `BLOCK_Y` = 1 and 24 threads at 5695.5 MFLOP/s, which is similar to the bigger matrix. Overall, the implementation performs similar on both matrix sizes; however, the performance numbers seem a bit more regular on bigger matrices, which could also be due to the fact that caching effects are more prevalent on bigger data sets. A bigger matrix size could not be run on Tsubame because the job submission system killed such tasks due to excessive memory usage.

When looking at specific configurations, their performance can often be explained with caching effects. For example, the best configuration with 30 threads on the small matrix is `BLOCK_X` = 32 and `BLOCK_Y` = 64. With 30 threads, every thread processes a range of 4096/30 = 135 elements in Y direction. With a blocking factor of 32 in X direction, this results in blocks of size 135 × 32, each of which has a size of 17 KB. Blocks of this size fit into the L1 cache, whereas this is no longer the case for bigger X blocking of 64 or more.

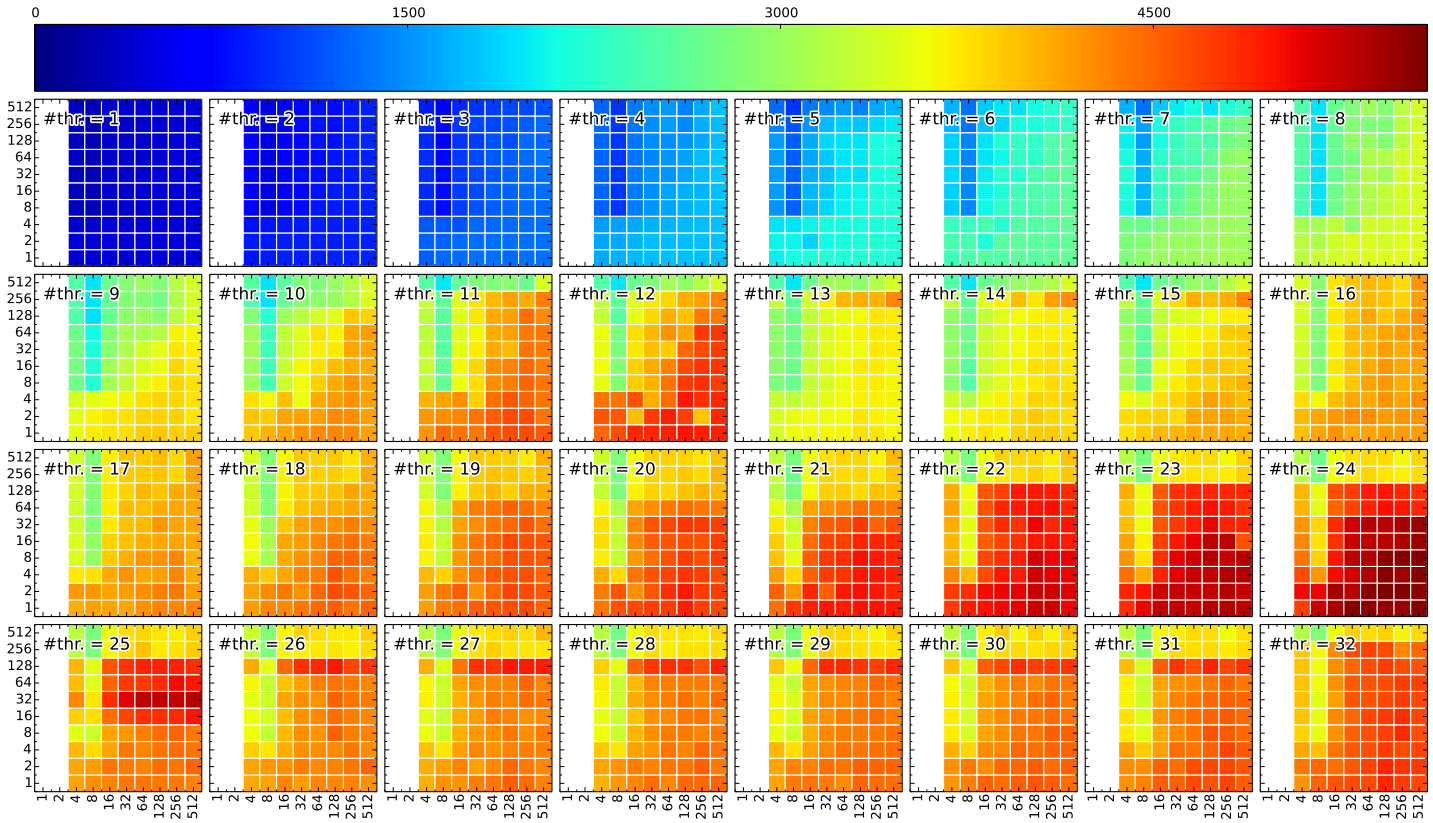


Figure 6: Performance Overview with Vectorization: All blocking combinations (powers of 2) in X and Y direction.

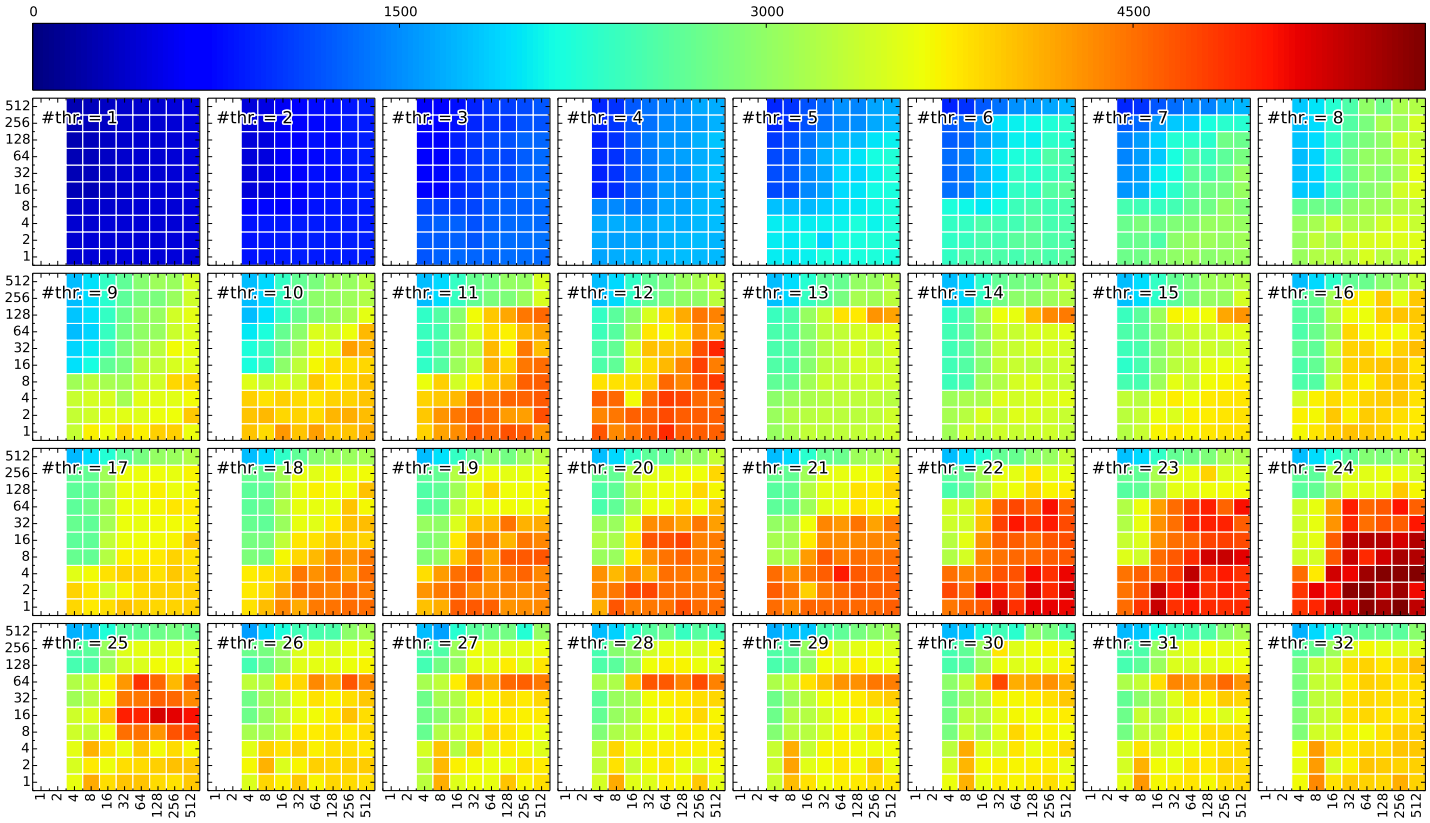
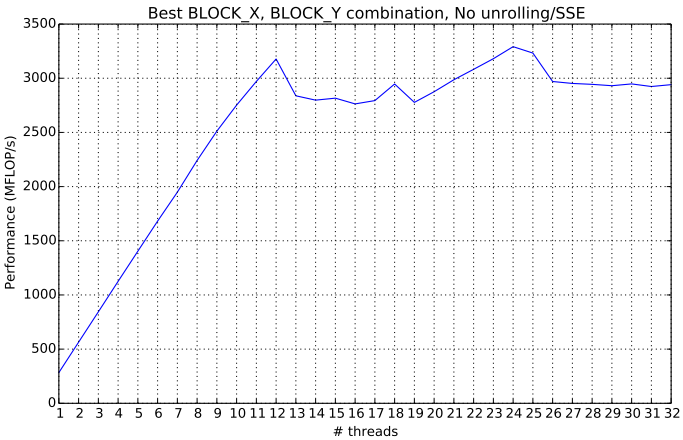
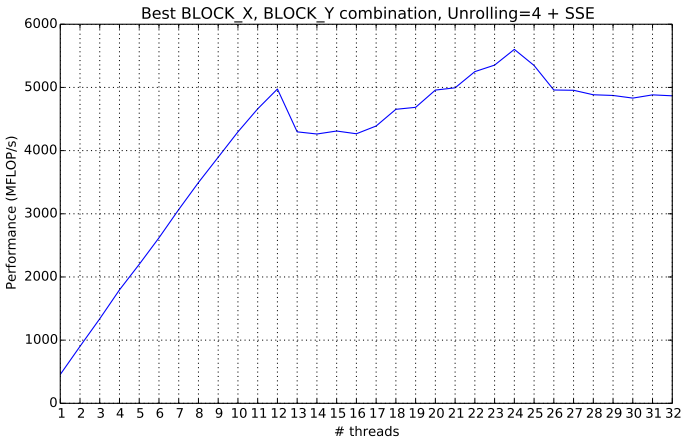


Figure 7: Performance Overview with Vectorization (Small Matrix)

Finally, Figure 8 compares the overall performance of the implementation without any unrolling or vectorization with the performance of the implementation where X is unrolled for 4 iterations and those 4 operations are vectorized on a big matrix (8194×8194). For every number of threads, the best combination of blocking factors in X and Y direction is chosen. The graphs shows that those optimizations always pay off.



(a) No Unrolling / Vectorization



(b) Unrolling 4 Iterations of X and Vectorization

Figure 8: Performance Overview with/without Unrolling and Vectorization: For every number of threads, the best blocking combination (X and Y) is chosen.