

Optimization and Parallelization of Ptychography Reconstruction Code

Thomas L. Falch¹ Jostein B. Fløystad² Anne C. Elster¹
Dag W. Breiby²

¹Department of Info. and Computer Science ²Department of Physics

Norwegian University of Science and Technology

{thomafal, elster}@idi.ntnu.no

{jostein.floystad, dag.breiby}@ntnu.no

Abstract

Ptychography is a promising new microscopy technique based on numerical reconstruction of coherent scattering patterns. The reconstruction of the real space image from the experimentally obtained scattering patterns requires substantial computational effort. In this paper we describe how to optimize and parallelize a ptychography reconstruction program that is using the difference map algorithm. Starting with a Python version, we translate it to C, and adapt it to take advantage of modern multi-core processors. The code is parallelized and optimized using thread-pools and SIMD instructions, as well as array merging, expression rewriting and loop unrolling. On a modern 4-core processor with simultaneous multithreading, we are able to achieve a speedup by a factor 14.2 over the initial Python version, and 6.5 over the initial, serial C version. This is a significant result, essentially turning the analysis of the X-ray microscopy results into an interactive rather than offline job. This paper provides several useful hints for users optimizing and parallelizing other kinds of scientific code.

1 Introduction

Ptychography is a promising new microscopy technique. In contrast to other microscopies, the data from a ptychography experiment is a set of far-field diffraction patterns, rather than images of the object studied. Reconstructing an image of the object from this data requires a significant computational effort. As is often the case in experimental practice, the images resulting from one experiment is used to determine what to do next. Clearly, having to wait for a long time for each image "to develop" makes such an interactive workflow impractical. A reasonable target for the performance of the reconstruction implementation, is that it should be as fast as the data acquisition or faster.

The "power wall" has made further scaling of processor frequency impractical. However, the continued reduction of transistor size, combined with the need for, and

This paper was presented at the NIK-2012 conference; see <http://www.nik.no/>.

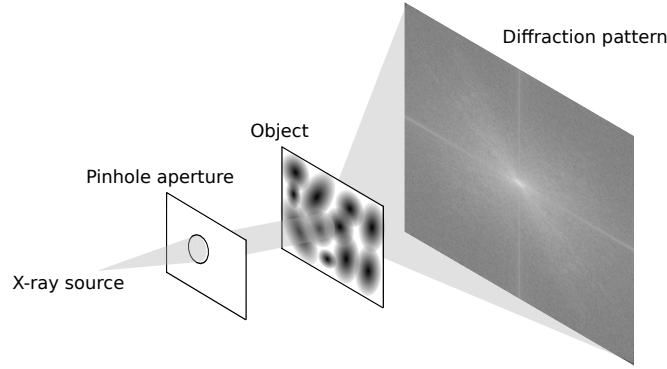


Figure 1: Diffractive X-ray microscopy setup. Coherent X-rays are used to illuminate a sample, causing a diffraction pattern to be formed, from which the real space image can be obtained by numerical reconstruction.

expectation of, ever increasing performance has in recent years led to a proliferation of parallel hardware, including systems with multiple processing cores on a single chip. It is well known, however, that parallel programming is much more difficult than serial programming.

In this paper, we show how a program performing ptychographic reconstruction can be adapted to take advantage of parallel hardware, and thus significantly improve its performance, essentially changing the reconstruction from an offline to an interactive job. In particular, starting with an unoptimized, serial program, we show how it can be parallelized and optimized for a modern multi-core processor.

2 Ptychography

The wavelength of visible light (400nm – 800nm) puts an upper limit on the resolving power of traditional optical microscopes. To study objects with finer detail, radiation with shorter wavelengths such as X-rays (wavelengths ~ 0.1 nm) and the matter waves of accelerated electrons (wavelengths ~ 0.002 nm) is required. Although lenses exist for both X-rays and electrons, they have problems with imperfections or ineffectiveness. Ptychography is a microscopy technique which avoids these problems by not using lenses, relying instead on computers to reconstruct the image of the object from recorded diffraction patterns. This technique has been implemented for both X-rays[1] and electrons[2].

A schematic of the experimental setup for ptychography is shown in Figure 1. A detector records the diffraction pattern formed while a coherent wavefield of finite spatial extent, referred to as the *probe*, is used to illuminate the object. The diffraction pattern measured is the absolute square of the Fourier transform of the product of object and probe. Since the detector does not record the phase information of the complex-valued wavefield, the object cannot be retrieved by a simple inverse Fourier transform. The main idea of ptychography is to record diffraction patterns at different positions of different, but overlapping regions of the object. Because of this overlap, the information about how a certain region of the object scatters the waves is recorded several times, as illustrated in Figure 2.

The retrieved image must satisfy two constraints: (a) each measured diffraction pattern must match the absolute square of the Fourier transform of the product of object

and probe, and (b) where a part of the object has been illuminated several times, the reconstructed object from the different diffraction patterns must agree. Algorithms based on alternating between applying these two constraints in an iterative procedure have been shown to solve the problem in a robust and efficient manner[3, 4, 5].

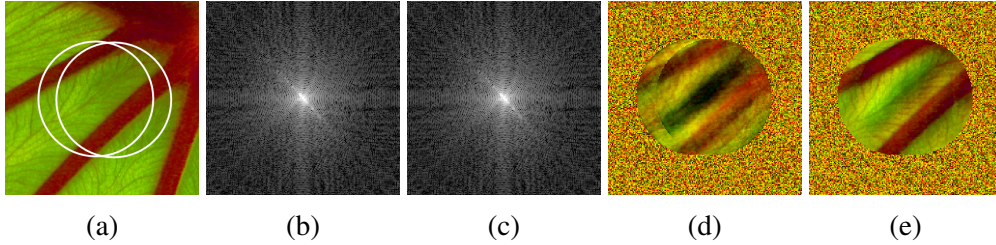


Figure 2: Illustration of ptychographical reconstruction. (a) The image, here assumed to be known for the sake of illustration, with indications of two partially overlapping circular areas that have been illuminated. (b) and (c) show the slightly different diffraction patterns corresponding to the two positions indicated in (a). (d) and (e) show that the algorithm reconstructs the object, after 3 (d) and 100 (e) iterations.

The difference map algorithm for ptychography yields an image of the object based on the measured diffraction patterns for each position and an initial guess for the complex-valued probe. The main steps can be summarized as follows:

1. Initialize the object with random values.
2. Find the Fourier transform of the product of the object and the probe function at the current position.
3. Update this Fourier transform by keeping the calculated phases, and replacing the amplitudes with those actually measured for the current position.
4. Find the inverse Fourier transform of the current wavefield, and use it to update the object in the area corresponding to the current position.
5. Update the probe by essentially setting it to the updated wavefield divided by the newly updated object.
6. Repeat steps 2-5 for each position.
7. Combine the results for all the positions
8. Repeat steps 2-7 for a predefined number of iterations.

Typically, 200-300 iterations are used to reach an essentially converged solution using X-ray data. A reasonable image is often obtained after about 100 iterations or less, depending on the contrast of the object. Figure 2 illustrates how the algorithm gradually reconstructs the object. Acquiring data for one image takes a few minutes, scaling linearly with the area of the field of view.

3 Optimization and Parallelization

Starting with an unoptimized Python program for ptychographical reconstruction, we translated it into C, and subsequently parallelized and optimized it. Here, we will describe how this was done.

The code snippets presented in the following sections have been simplified for clarity¹.

Parallelization

In this section, we will describe how we parallelized the program using coarse-grained parallelization of `for` loops, and then more fine grained parallelization using (single instruction, multiple data) SIMD instructions.

Thread Pools

Inspection of the algorithm revealed several triply nested `for` loops that are obvious candidates for parallelization. An example of one such triply-nested loop is shown in Listing 1.

The parallelization can be implemented with a thread pool. At the beginning of each loop, the master thread divides the iterations evenly between the worker threads, and when they are finished, collects and combines their results. Despite the apparent simplicity of this scheme, some complications arise.

Load balancing is important for optimal performance. If some threads finish significantly earlier than others, resources might be wasted. Even if we have the same number of threads as physical cores, and assign them the same amount of work, they might not finish at the same time. This can be caused by some threads being preempted by the operating system so that it can perform its work. Furthermore, since the threads share resources such as main memory, inter-thread interference might, during a single execution, cause some threads to be slowed down relative to others.

To mitigate this problem, we have implemented a system for dynamic load balancing. Rather than dividing the work statically between the threads, we create a queue of work items (in this case loop iterations). Each thread will remove an item from the queue, execute it, and return for more until the queue is empty.

The optimal size for the work items is not obvious. Smaller items will ensure more even load balancing, but also cause more overhead, as the shared queue is accessed more frequently. The solution we have adopted, inspired by *work stealing* [6], combines small and large work items to get the advantages of both. We let the first item be fairly large, but small enough to make sure all threads will finish it even if slowed down. Subsequent items are much smaller to evenly distribute the work.

Loop Organization

As previously noted, the `for` loops we are parallelizing are triply nested. For each exposure, we access all the rows and columns of the 2D array representation of the discretized complex-valued probe, and of the 2D array representing the complex-valued object in the area the probe overlaps for that particular exposure. While we later describe how some nested loops can be flattened, this can not be done for these loops, since we, for each exposure, overlap the object with the smaller probe in a different area.

In one of the instances of a triple `for` loop, it was clearly advantageous from a code-structuring point of view to let the loop over exposures be the outermost, since we are

¹The full source code is freely available at <http://idi.ntnu.no/~thomafal/kode>.

```

for(int exp = 0; exp < no_of_exps; exp++){
    base_row = get_base_row(exp); base_col = get_base_col(exp);

    for(int row = 0; row < probe_width; row++){
        for(int col = 0; col < probe_width; col++){
            obj_row = base_row + row; obj_col = base_col + col;

            float real = creal(object[obj_row][obj_col]);
            float imag = cimag(object[obj_row][obj_col]);
            probe_denominator[row][col] += real*real + imag*imag;

            float complex c = conj(object[obj_row][obj_col]);
            new_probe[row][col] += iterate[exp][row][col]*c;
        }
    }
}

```

Listing 1: Before adding precomputation. In each iteration, we compute the squared absolute of the object array at the given position.

taking the forward and inverse Fourier transform of each exposure. For other cases of triple `for` loops, we have no such incentives. This leads to three main alternatives, as one can let either the exposure, the row or the column loop be the outermost loop, which one parallelizes. While the amount of work to be done is invariant, the memory access patterns, and more importantly, the cache friendliness of these, might vary widely. We have implemented all three strategies, and measured their relative performance.

SIMD Instructions

In several loops, we perform the exact same operation on every item of one or more arrays. Here we can use SIMD instructions [7], which operate on several neighbouring array items in parallel. We use 128 bit wide SIMD instructions, allowing us to operate on two complex or four real single precision floating point numbers in parallel, thus reducing the number of iterations of the loops by a factor of two or four, respectively. We added the SIMD instructions manually using compiler intrinsics.

Optimizations

In this section, we will look at the various optimization techniques we used.

Precomputation

As shown in Listing 1, the square of the absolute value of the complex numbers in the object array are computed. We do not access the same area of the object array in each iteration, but there is a significant overlap, which implies redundant and hence unnecessary computations.

Since this is an expensive operation, it might be beneficial to precompute it. At the same time as the object array is updated, we can create an additional array containing the squared absolute of the object array, and access this precomputed array in the loop. Note, however, that while precomputation reduces the total amount of computation, it increases memory traffic, and might therefore not be any faster.

Array Merging

Array merging is a technique aimed at increasing the spatial locality of the data access pattern [8]. When accessing multiple arrays in a loop, we can reorganize the data into one

array, containing structs with the previous arrays elements as members. That way, all the memory accesses of a single iteration will go to the same area in memory. Listings 2 and 3 illustrate how this could be done in the present case.

```
for(int row = 0; row < probe_height; row++){
    for(int col = 0; col < probe_width; col++){
        i++;
        probe[i] = new_probe[i] / probe_denominator[i];
    }
}
```

Listing 2: Before array merging. In each iteration of the loop, the i 'th elements of `new_probe` and `probe_denominator` are accessed. These are at widely different locations in memory.

```
typedef struct{
    float complex new_probe;
    double probe_denominator;
} probe_update_t;

for(int row = 0; row < probe_height; row++){
    for(int col = 0; col < probe_width; col++){
        i++;
        probe[i] = b[i].new_probe / b[i].probe_denominator;
    }
}
```

Listing 3: After array merging. `b` is an array of structs. The two locations accessed in this array are close to each other in memory.

Moving Initialization

In Listing 1 we see how array values are updated in one triply nested loop in the code. These arrays must be reset to all zeros each time the loop is entered. Rather than doing this for the entire arrays prior to the loop, we move the initialization into the loop. Immediately before we access an array element for the first time, we set it to zero. The increased complexity of the necessary additional conditional branch is compensated for by increased spatial and temporal locality of the memory access pattern.

Changing Branch Condition

```
//Original branch
if(item.start == exp && item.first == 1){
    probe_denominator[row][col] = 0;
    new_probe[row][col] = 0 + 0*I;
}
//Modified branch
if(item.first == 1){
    if(item.start == exp){
        probe_denominator[row][col] = 0;
        new_probe[row][col] = 0 + 0*I;
    }
}
```

Listing 4: A `if` branch in the body of a triply nested loop, original version on top, modified below.

Listing 4 shows the conditional branch in the body of a triply nested loop. As previously described, the outermost of these loops are parallelized. Each thread will execute several "chunks" of iterations. For instance, one thread may execute iterations 10-20, 35-45 and 110-111. The size and number of chunks may vary, and the chunks not executed by one thread will be executed by others.

Clearly, a thread needs to set the buffers to zero on the first iteration, of the first chunk. A chunk is described by the `item` struct. The chunk consists of the iterations from `item.start` to `item.stop`. Further, if `item.first` equals 1, this is the first chunk, which explains the condition in the original branch.

To make this branch easier to predict, we split it into two branches. The `item.start == exp` condition will be true on the first iteration of every chunk, and since a thread might execute many chunks, it is hard to predict the result of the condition. The value of `item.first` is, however, only 1 once for each thread, and then 0 in the remaining iterations and chunks, and is therefore easier to predict.

Expression Rewriting

The latency of floating point divisions are about 2-10 times higher than floating point multiplication[9]. It is therefore useful to rewrite expressions so that they use multiplication, rather than division, if possible. For example, in one place in the code we compute an expression with the following form:

$$a = b \frac{\frac{c}{d}}{e + f}.$$

This can be rewritten as

$$a = b \frac{c}{d(e + f)},$$

where a division has been replaced with a multiplication.

Array Layout

We use several arrays which represents two dimensional functions (i.e. the `probe` and `object` arrays), in the program. It is therefore natural to use two dimensional arrays, and access the elements using row and column indices. While this representation makes sense from a physics readability point of view, it might be detrimental to performance. The code in Listing 5 provides an example.

```
//Original loop
for(int row = 0; row < probe_height; row++){
    for(int col = 0; col < probe_width; col++){
        f_abs_dev[row][col] = f_abs[row][col] - m_amps[i][row][col];
    }
}
//Modified loop
for(int index = 0; index < probe_size; index++){
    f_abs_dev[index] = f_abs[index] - m_amps[i*probe_size + index];
}
```

Listing 5: The original loop, on top, and the modified, below. Even though `f_abs_dev` and `f_abs` represent 2D functions, and `m_amps` represents a 3D function, it is not necessary to use 2D and 3D arrays to represent them.

Two dimensional arrays are either implemented as one contiguous area in memory or as a array of pointers, where each pointer points to a separate row. In the first

case, when `f_abs[row][col]` is accessed, the offset into this area, which is `row*probe_width+col` has to be calculated. In the second, we must perform expensive pointer chasing, and also have the problem that consecutive rows might not be adjacent in memory, resulting poorer cache prefetching performance. Furthermore, removing the inner loop eliminates one comparison and increment on every iteration. More advanced related techniques can be found in [10].

Loop Unrolling

Changing array layout as described above, is to a certain degree a special case of a more general optimization technique known as *loop unrolling*. There are many variations of the technique, but in the simplest case, a loop is simply replaced with all the instructions it performs listed sequentially. As an example, a `for` loop adding the elements of two 5 element arrays is replaced with five instructions doing this explicitly. Loop unrolling improves performance in two ways. Firstly, the comparison every iteration is removed. Secondly, branch misprediction when exiting the loop is avoided.

Loop unrolling can be performed by hand, which is often tedious, and might cause the code to become more unreadable. More conveniently, the compiler can often perform loop unrolling for us, if the number of iterations is known at compile time. With GCC, this can be done using the `-funroll-loops` argument.

4 Results and Discussion

Our experiments were conducted on a desktop workstation with an Intel Core i7 930 4-core processor. Further details about hardware and software used can be found in Table 1. Unless otherwise noted, eight threads were used in all tests to take advantage of hyperthreading. When comparing our optimized version to the Python version, real X-ray data were used. When comparing the different optimization techniques, synthetic data were used.

Software		Hardware	
Component	Version	Component	Details
Ubuntu	4.11	Processor	Intel Core i7 930
Linux kernel	2.6.38-11	Processor frequency	2.8 GHz
GCC	4.4.5	Physical processor cores	4
Python	2.7.1	Locical processor cores	8
		Cache	4 x 256 KB L2, 8 MB L3
		Main memory	12GB

Table 1: Details of experimental setup.

The net result of our efforts is shown in Figure 3, which shows the speedup of our final version compared with the original Python code, and our initial serial C code, respectively. These results are as expected, as it is well known that C is significantly faster than Python. The speedup achieved compared to the serial C version is also as anticipated, taking into account that we are using four simultaneous multithreading-enabled physical cores, as well as SIMD instructions, and several optimizations. In absolute terms, with 200 iterations, this corresponds to a reduction from about 420 s to about 30 s for a real dataset. As a reference, acquiring the data for this experiment takes about 150 s.

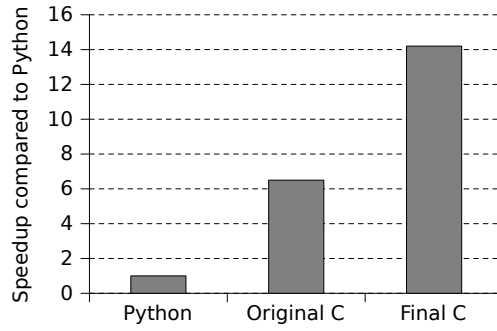


Figure 3: The speedup of the optimized, parallelized code compared to the original serial C and Python versions.

Parallelization

Our load balancing approach only gave noticeable speedups when more than five threads were used, compared to naively dividing the work evenly between the threads. With eight threads, the naive approach was 3% slower than our approach.

While the results of using the exposure and row loops as the outermost were about equal, letting the column loop be the outermost resulted in significantly deteriorated performance, but only when more than five threads were used. With eight threads, using the column loop as the outermost were about 78% slower than row and exposure. This difference is caused by more unfavourable memory access patterns.

Using SIMD instructions, as discussed in Section 3, reduces the running time by 34%, corresponding to a speedup of 1.5. We can apply SIMD instructions in functions originally accounting for approximately 70% of the running time. Since we are mostly using double precision real numbers, or single precision complex numbers, SIMD instructions should give a speedup of about a factor 2. Using Amdahl's law, with p being the parallel fraction, and N the number of processors, we get:

$$S_{predicted} = \frac{1}{(1-p) + \frac{p}{N}} = \frac{1}{0.3 + 0.7/2} = 1.54$$

This fits nicely with our observed results.

Scalability

Figure 4 shows speedup and efficiency for 1-8 threads, with and without SIMD instructions. The speedups are good for 2-4 threads, but remain fairly constant or even falls once we start using hyper-threading. We see the same trend for the efficiencies.

The speedup and efficiency are better when not using SIMD. This is because SIMD is used only in the parallelizable parts, so with one thread, less time is spent in the parallel section. In Amdahl's terms, its parallel fraction is smaller. It is then clear that the achievable speedup when using more threads is lower. While hyper-threading can improve performance, it is not equivalent to adding more physical cores, and the lack of speedup is therefore anticipated [11].

Figure 5 shows the result of fitting Amdahl's law to the speedups observed for the final version of the program. We only use data for one to four threads, as the assumptions of Amdahl's law fit poorly with simultaneous multithreading. As we can see, the observed speedups fit almost perfectly with Amdahl's law using $p = 0.89$. The figure also shows

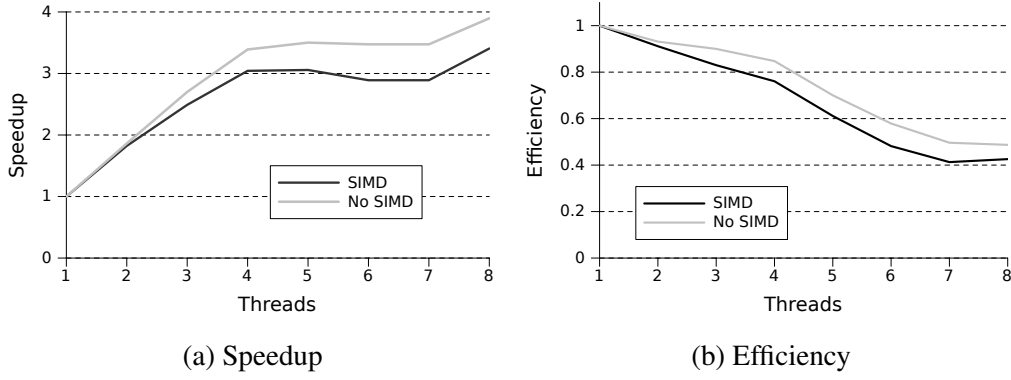


Figure 4: Speedup (a) and efficiency (b) for one through eight threads, with and without SIMD instructions.

the predicted speedups for five and six threads when using five and six physical cores. This result shows that, despite its simplicity, Amdahl's law can still be used in practice.

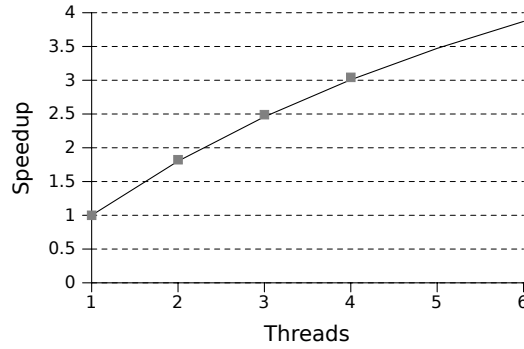


Figure 5: The result of fitting Amdahl's law to the observed speedups. The squares indicate the observed speedups. The line is Amdahl's law, with parameters best fitting the data.

Optimizations

The results of our optimization efforts are shown in Figure 6 where the optimizations described above are successively applied. This figure shows how applying each new optimization improves performance, with the exception of precomputation. However, precomputation actually improves performance when SIMD instructions are not used, as we elaborate upon below.

More detailed profiling results and discussion of them can be found in [12].

The precomputation optimization technique yields interesting results. When we are not using SIMD instructions, precomputation pays off. However, when SIMD instructions are used, the result is reversed: precomputation results in poorer performance. This is illustrated in Figure 7. These results can be explained by recalling that precomputation increases the number of cache misses because we must access the precomputed values. When we are not using SIMD instructions, this does nonetheless increase performance, because we avoid unnecessary and expensive floating point computations.

When we are using SIMD instructions, we dramatically increase the number of FLOPS the processor can perform, in other words, we are making floating point

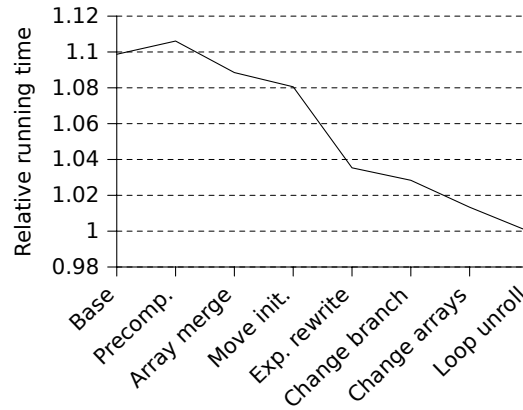


Figure 6: Timing results for the techniques discussed in this section. We show the relative slowdown compared to the best result shown here. As we see the performance steadily improves, with the exception of precomputation, see below.

computation cheaper. In this case, computation becomes so cheap, that it is better to perform repeated (unnecessary) computations than having to load values from memory. With SIMD instructions, we are, in essence, transforming the originally compute bound problem into one that is memory bound.

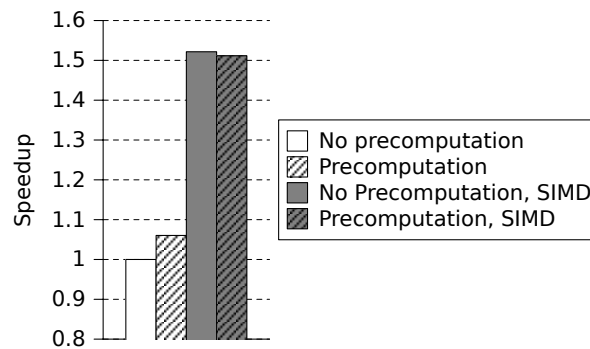


Figure 7: Precomputation with and without SIMD. SIMD is seen to significantly reduce the compute time. While using precomputation significantly increases performance when *not* using SIMD instructions, it slightly decreases performance when using them.

5 Conclusion

In this paper, we have discussed how the performance of a program performing ptychographical reconstruction can be improved. An existing program, written in Python was first translated to C, and subsequently parallelized and optimized with various techniques. Specifically, we parallelized several time-consuming `for` loops using thread pools. We also parallelized the bodies of those and other loops using SIMD instructions. The optimizations we applied were designed using general knowledge of hardware architecture to allow for a more efficient use of the resources, and included array merging, expression rewriting and loop unrolling.

With the exception of precomputation, all the optimization techniques tested decreased the running time of the program. The parallelization were responsible for the most significant improvements in performance. Overall, the final version was 14.2 times

faster than the original Python program, and 6.5 times faster than the unoptimized, serial C version. This is a significant result, essentially turning the reconstruction of ptychography data from an offline to an interactive job, opening both for more efficient exploitation of scarce beamtime at synchrotrons and, ultimately, improved experiments.

Possibilities for future research include autotuning[13], and porting the application to GPU-systems.

References

- [1] J. M. Rodenburg *et al.*, “Hard-x-ray lensless imaging of extended objects,” *Physical Review Letters*, vol. 98, p. 034801, Jan. 2007.
- [2] M. Humphry *et al.*, “Ptychographic electron microscopy using high-angle dark-field scattering for sub-nanometre resolution imaging,” *Nature Communications*, vol. 3, p. 730, Mar. 2012.
- [3] V. Elser, “Phase retrieval by iterated projections,” *Journal of the Optical Society of America A*, vol. 20, pp. 40–55, Jan. 2003.
- [4] P. Thibault *et al.*, “Probe retrieval in ptychographic coherent diffractive imaging,” *Ultramicroscopy*, vol. 109, pp. 338–343, Mar. 2009.
- [5] A. M. Maiden and J. M. Rodenburg, “An improved ptychographical phase retrieval algorithm for diffractive imaging,” *Ultramicroscopy*, vol. 109, pp. 1256–1262, Sept. 2009.
- [6] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM*, vol. 46, no. 5, 1999.
- [7] Intel, *Intel(R) 64 and IA-32 Architectures Developer’s Manual*.
- [8] C. Pancratow *et al.*, “Why computer architecture matters: Memory access,” *Computing in Science and Engineering*, vol. 10, no. 4, 2008.
- [9] C. Pancratov *et al.*, “Why computer architecture matters,” *Computing in Science and Engineering*, vol. 10, no. 3, 2008.
- [10] A. C. Elster, *Parallelization issues and particle-in-cell codes*. PhD thesis, Cornell University, 1994.
- [11] D. T. M. D. Koufaty, “Hypertreading technology in the netburst microarchitecture,” *Micro, IEEE*, vol. 23, no. 2, 2003.
- [12] T. L. Falch, “Optimization and parallelization of ptychography reconstruction code,” Project Report, Department of Computer and Info. Science, Norwegian University of Science and Technology, 2011.
- [13] A. C. Elster and J. C. Meyer, “A super-efficient adaptable bit-reversal algorithm for multithreaded architectures,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –8, May 2009.