Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 479–488
DOI: 10.15439/2022F251
ISSN 2300-5963 ACSIS, Vol. 30

Influence of loop transformations on performance and energy consumption of the multithreded WZ factorization

Beata Bylina, Jarosław Bylina, Monika Piekarz
Institute of Computer Science, Marie Curie-Sklodowska University
Pl. M. Curie-Skłodowskiej 5
Lublin, 20-031, Poland
Email: {beata.bylina, jaroslaw.bylina, monika.piekarz}@umcs.pl

***Abstract*—High-level loop transformations are a key instrument to effectively exploit the resource in modern architectures. Energy consumption on multi-core architectures is one of the major issues connected with high-performance computing. We examine the impact of four loop transformation strategies on performance and energy consumption. The investigated strategies include: loop fission, loop interchange (permutation), strip-mining, and loop tiling. Additionally, a column-wise and row-wise store formats for dense matrices are considered. Parallelization and vectorization are implemented using OpenMP directives. As a test, the WZ factorization algorithm is used. The comparison of selected strategies of the loop transformation is done for Intel architecture, namely Cascade Lake. It has been shown that for WZ factorization, which is an example of an application in which we can use the loop transformation, optimization towards high-performance can also be an effective strategy for improving energy efficiency. Our results show also that block size selection in loop tilling has a significant impact on energy consumption.**

Keywords: energy saving, energy consumption, RAPL, WZ factorization, multicore architecture

I. Introduction

With the growing demand for high-performance computing, new architectures have emerged which unfortunately consumes more and more energy. Reducing energy consumption in these architectures is one of the major challenges. The current research trends based on performance studies [12], [13], [16] and comparisons are to develop hardware and software to achieve the best performance and energy compromise. One of the aspects of creating energy-aware software is the optimization of implementation of complex numerical algorithms. Such complex algorithms include loops, in particular nested ones. Nested loops are an important structure bearing a great deal of the parallelism and vectorization possibilities. However, to parallelize them efficiently, the programmer has to make some decisions about applying various transformations. An example of such loops is matrix algorithms, like matrix multiplication or different kinds of factorizations, widely investigated in the literature [3], [1].

In the work [2], we studied loop transformations for nested loops on multicore architectures on the example of a factorization similar to the LU factorization, namely, the WZ factorization [19]. The WZ factorization has some nontrivial data dependencies and the compiler is not able to efficiently optimize the algorithm. We have chosen the following four: loop fission, loop interchange (loop permutation), strip-mining, and loop tiling.

In this article, we investigate the impact of these four loop transformations on performance and energy consumption for the WZ factorization on multicore architecture. We describe in detail two block-related transformations (strip-mining and loop tiling). We are making theoretical and experimental considerations about the size of the blocks. Additionally, we consider column-wise and row-wise storage formats for dense matrices. The OpenMP standard is used for parallelization and vectorization of the code. The Intel RAPL (Running Average Power Limit) [7], [9] interface is used as a source of information on energy consumption.

The main contributions of this article are the following:

- results of the tests from the evaluation of the execution time and energy consumption for four loop transformations of the WZ factorization for various data sizes on Intel architecture — namely, Cascade Lake;
- conclusions on the impact of the four loop transformations on the execution time and energy consumption;
- analysis of the correlation between performance and energy consumption for four loop transformations.

This paper is organized as follows. Section II presents a few related works on loop transformations and energy consumption/performance analysis for modern computer architectures and systems. Section III discusses the four loop transformations applied here for the WZ factorization and studies block size for tiled transformation. In Section IV, we concentrate on the details of conducting tests and on the discussion and explanation of the results. In Section V, we present the conclusions of the numerical experiments and further research directions.

II. Related works

When high-performance computing is considered, energy consumption is one of the most important challenges. These challenges are analyzed on many levels. In particular, the works of [11], [13], [16] dealt with the topic of energy

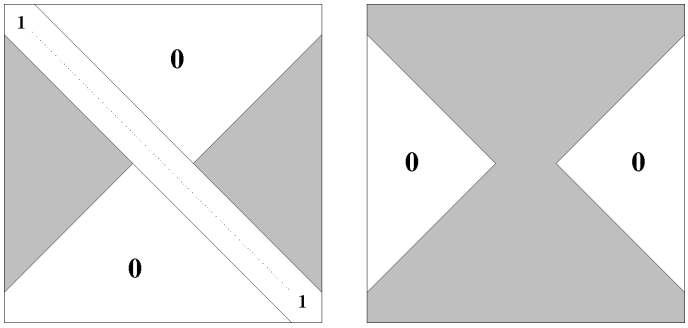


Fig. 1: The output of the WZ factorization — forms of the matrices **W** (left) and **Z** (right).

consumption in the context of using the OpenMP standard for multi-core computers with shared memory.

The key to software optimization in terms of the performance for algorithms that include nested loops is the right choice of the appropriate loop transformations. Loop transformations are a research topic for various automatic optimization techniques [8], [10], [14], [15] as well as for manual conversion of application code [5], [17], [18] so as to obtain the best possible performance on modern multi-core architectures.

In many numerical algorithms where dependencies between data are very complicated, even such tools as efficient optimizing compilers are not able to transform the code to use the potential of modern processors. The authors of [3], [1], present algorithms for solving systems of equations, trying to improve their performance, in particular in parallel. Improvement in performance was obtained by appropriate transformation of the underlying algorithm using looping tiling and appropriate data structures. There are currently not too many studies in the literature about both efficiency and energy consumption in the context of loop transformations. In our work, we study a numerical algorithm (the WZ factorization), in which, loops are transformed. This algorithm concerns numerical linear algebra, in particular solving systems of equations on multi-core architectures using OpenMP in the context of performance and energy consumption.

III. WZ FACTORIZATION

The WZ factorization (Figure 1) was introduced in [4]. It was a new method for parallel solving of systems of linear equations on computers containing many data processing units and it was an alternative to the well-known LU factorization.

The WZ factorization is based on the decomposition of the square matrix **A** into two matrices: **W** i **Z**. All the matrices which we consider are dense ones. They are stored as one-dimensional arrays in one of the two formats: column-wise or row-wise.

The basic algorithm for the WZ factorization for an even size of the matrix (we only consider even sizes — without loss of generality) is shown in Figure 2.

The loop transformation consists in replacing itself with an equivalent loop containing the structured block. Some

```
for(k = 0; k < n/2-1; k++) {
    p = n-k-1;
    akk = a[k][k];    akp = a[k][p];
    apk = a[p][k];    app = a[p][p];
    detinv = 1 / (apk*akp - akk*app);
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p] - app*a[i][k])
                  * detinv;
        w[i][p] = (akp*a[i][k] - akk*a[i][p])
                  * detinv;
        for(j = k+1; j < p; j++)
            a[i][j] = a[i][j]
                      - w[i][k]*a[k][j]
                      - w[i][p]*a[p][j];
    }
}
```

Fig. 2: The basic algorithm for the WZ factorization — pseudocode.

```
for(k = 0; k < n/2-1; k++) {
    . . .
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p] - app*a[i][k])
                  * detinv;
        w[i][p] = (akp*a[i][k] - akk*a[i][p])
                  * detinv;
    }
    for(i = k+1; i < p; i++)
        for(j = k+1; j < p; j++)
            a[i][j] = a[i][j]
                      - w[i][k]*a[k][j]
                      - w[i][p]*a[p][j];
}
```

Fig. 3: The algorithm after the fission of the `i`-loop — pseudocode. This algorithm matches the row-wise layout.

well-known transformations considered are: loop fission, loop interchange (permutation), strip-mining, tiling.

A. Loop fission and permutation

Loop fission (also called loop distribution) breaks a loop into multiple loops over the same index range but each taking only a part of the loop's body. Its purpose is to achieve better utilization of locality of reference — isolate parallelizable loops create independent loops, hence creating separate tasks Its result is shown in Figure 3.

In the fission algorithm (Figure 3) it is possible to use the loop interchange (for the `j`-loop and the second `i`-loop). The loop interchange transformation switches the order of loops' nesting (consists in replacing the internal loop with the external one). The purpose of such a transformation is to improve data locality or increase parallelism and vectorization. The algorithm from Figure 3 is denoted as `fission-ij`. Its result is shown in Figure 4.

```
for(k = 0; k < n/2-1; k++) {
    . . .
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p] - app*a[i][k])
                  * detinv;
        w[i][p] = (akp*a[i][k] - akk*a[i][p])
                  * detinv;
    }
    for(j = k+1; j < p; j++)
        for(i = k+1; i < p; i++)
            a[i][j] = a[i][j]
                      - w[i][k]*a[k][j]
                      - w[i][p]*a[p][j];
}
```

Fig. 4: The algorithm after the fission of the `i`-loop with permutation loop — pseudocode. This algorithm matches the row-wise layout.

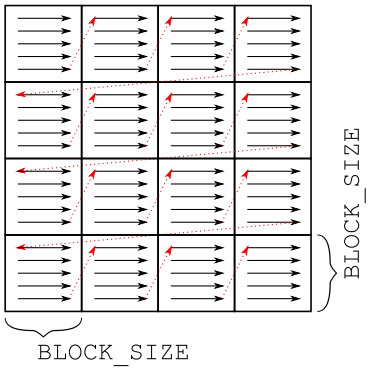


Fig. 5: Computing sequence after loop tiling (black: computations in blocks; red: order between blocks).

B. Strip-mining and loop tiling

Access to the main memory in our algorithm takes a lot of time. It is a well-known fact that the cost of accessing the memory is much higher than the cost of computations. We can even figure the number of memory reads and writes (C_M) and compare it to the number of floating-point operations (C_F). After some simple calculations we obtain:

$$C_M = \frac{7}{6}n^3 + O(n^2),$$

$$C_F = \frac{2}{3}n^3 + O(n^2).$$

Thus, the ratio of memory access to computations is:

$$\frac{C_M}{C_F} \approx \frac{7}{4}.$$

This means that we need a lot of memory access to perform our algorithm — almost two memory accesses for one floating-point operation. So, it is the main obstacle to utilizing the computing power of modern processors fully. A manner to solve this problem is to use the cache memory (which is much faster than the main memory) efficiently. Of course, the size of the cache memory is too small to house all the data needed in the algorithm.

Strip-mining is a loop transformation that consists in replacing one loop with two nested loops. One of them (inner) is appropriate for vectorization (it is quite short and with a unit stride), and another (outer) is longer and its step is equal to the full number of iterations in the inner one. The transformation pays only when the original loop is rather long.

A loop in the process of strip-mining is divided into two loops, where the inner one has `BLOCK_SIZE` iterations and the outer one has `n/BLOCK_SIZE` iterations (`n` is the number of iterations in the original loop). The strip-mining alone can have some positive impact on the performance (by easing the automatic vectorization process).

One of the widely used techniques which allow for improving performance is loop tiling which consists in connecting strip-mining with loop interchange. The main aim of this technique is to reduce the number of reads from and writes to the main memory by improving the spatial and temporal data locality, and hence by better utilizing the hierarchy of the memory — especially the cache memory by lessening the rate of cache misses. It is also useful for vectorization — both automatic and explicit — and for parallelization.

After such a transformation, the order of the computations changes (see Figure 5), although without the change of the result.

In such a process we improve the temporal and spatial locality of the data. By dividing the data into pieces of `BLOCK_SIZE`, we cause them to fit in cache memory (we mean level 1 cache here) and stay there as long as needed to conduct current computations. This minimizes the frequency of cache memory swaps. Too big `BLOCK_SIZE` and the data would not fit into the cache, too small `BLOCK_SIZE` and the swapping frequency rises. Moreover, to facilitate the vectorization, we wanted to make `BLOCK_SIZE` a multiple of the SIMD register. Unfortunately, in most of the iterations of both the `i`-loop and `j`-loop, the first and the last iteration is chopped. For this reason, we broadened the first and the last iteration to full `BLOCK_SIZE`. It does not change the results of the algorithm (additional operations are beyond the non-zero elements of the resulting matrices), although, it forces the machine to make some more computations (in Figure 6, the red color denotes the redundant computations) — the bigger `BLOCK_SIZE` the more additional computations are needed. The advantage is the possibility of vectorizing evenly all the iterations.

To achieve this, we make every outer loop start with a nearest full multiple of `BLOCK_SIZE` (rounded down) and we make every inner loop iterate through a whole `BLOCK_SIZE`. Moreover, all the matrices were allocated with the memory alignment suitable for the used architecture.

Figure 7 shows the original algorithm after parallelization, with strip-mining of the `j`-loop (the full loop tiling is impossible due to the infeasibility of the loop interchange). Figure 8 presents the fission algorithm with full loop tiling. The function `RDTTNM()` (stands for *round down to the nearest*

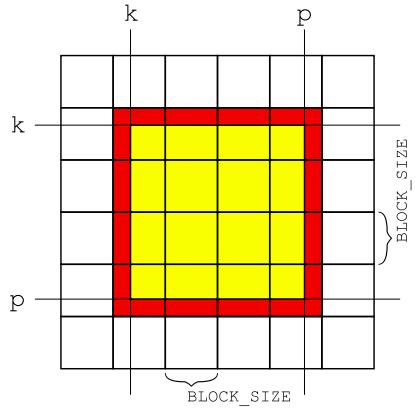


Fig. 6: Computations in the kth step of the algorithm after loop tiling (yellow: necessary computations; red: needless computations).

```
for(k = 0; k < n/2-1; k++) {
    . . .
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p] - app*a[i][k])
                  * detinv;
        w[i][p] = (akp*a[i][k] - akk*a[i][p])
                  * detinv;
        start = RDTTNM(k+1, BLOCK_SIZE);
        for(jj = start; jj < p;
                        jj += BLOCK_SIZE) {
            __assume(jj % BLOCK_SIZE == 0);
            for(j = jj; j < jj+BLOCK_SIZE;
                        ++j)
                a[i][j] = a[i][j]
                          - w[i][k]*a[k][j]
                          - w[i][p]*a[p][j];
...}
```

Fig. 7: Strip-mining in the basic algorithm — pseudocode.

multiple) can be defined as a macro:
`#define RDTTNM(a, r) (((a)/(r))*(r))`

In Figures 7 and 8, we use the compiler clause `__assume` which tells the compiler that a given condition is fulfilled — here, we declare that `ii` and `jj` are multiples of the `BLOCK_SIZE` which facilitates the vectorization.

Table I shows the computation overhead for the algorithm after loop tiling (red in Figure 6) — as a function of the size n of the matrix and of the `BLOCK_SIZE` (b).

Figure 9 shows the dependencies (black arrows) between the input data (green and blue dots) and the results (red dots) of the innermost loop in the kth step of the WZ factorization. We can see that — with no regard to the matrix memory layout (even for more complex layouts [6]) — the results of the computations depend on the data from various areas of the memory, so the loop tiling can give very limited performance improvements.

Here, the OpenMP standard is used for parallelization and vectorization code for all loop transformations. The more outer `i`-loop or `ii`-loop is parallelized with the pragma `parallel`

```
for(k = 0; k < n/2-1; k++) {
    . . .
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p] - app*a[i][k])
                  * detinv;
        w[i][p] = (akp*a[i][k] - akk*a[i][p])
                  * detinv;
    }
    start = RDTTNM(k+1, BLOCK_SIZE);
    for(ii = start; ii < p;
                    ii += BLOCK_SIZE) {
        for(jj = start; jj < p;
                        jj += BLOCK_SIZE) {
            __assume(ii % BLOCK_SIZE == 0);
            for(i = ii; i < ii+BLOCK_SIZE;
                        ++i) {
                __assume(
                    jj % BLOCK_SIZE == 0);
                for(j = jj; j < jj+BLOCK_SIZE;
                            ++j)
                    a[i][j] =
                          a[i][j]
                          - w[i][k]*a[k][j]
                          - w[i][p]*a[p][j];
...}
```

Fig. 8: Loop tiling in the fission algorithm — pseudocode.

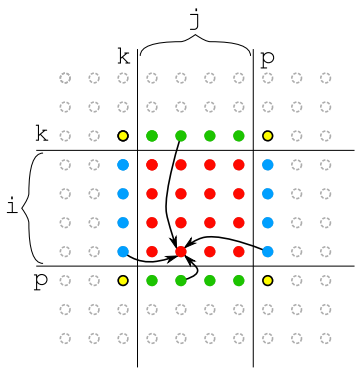


Fig. 9: The arrows show the data dependencies for updating an element of the array `a` in the kth step. Grey, green and yellow elements have their final values in the kth step; yellow ones are needed to compute `detinv` in this step; blue ones are elements of the array `w` computed in the middle loop of this step; red ones are updated in the innermost loop with the use of two blue elements and two green elements each.

`loop` and the most inner `j`-loop is vectorized with the pragma `simd` — details in [2].

IV. Numerical Experiment – Methodology and Results Analysis

A. Methodology

We test two types of versions of the WZ factorization algorithm:

- the basic algorithm presented in Figure 2 for the matrix stored in memory in two formats: a row-wise (`basic-row`) and column-wise (`basic-col`);

TABLE I: The ratio of the excess computations in the algorithm after loop tiling to the computations in the basic algorithm (n is the size of the matrix; b is the `BLOCK_SIZE`).

	$b =$ **8**	**16**	**32**	**64**	**128**	**256**	**512**
$n =$ **1024**	2.06%	4.44%	9.28%	19.24%	40.33%	87.21%	199.71%
2048	1.03%	2.21%	4.59%	9.42%	19.38%	40.48%	87.35%
3072	0.68%	1.47%	3.05%	6.24%	12.75%	26.29%	55.46%
4096	0.51%	1.10%	2.28%	4.66%	9.50%	19.46%	40.55%
5120	0.41%	0.88%	1.82%	3.72%	7.57%	15.44%	31.94%
6144	0.34%	0.73%	1.52%	3.10%	6.29%	12.80%	26.34%
7168	0.29%	0.63%	1.30%	2.65%	5.38%	10.93%	22.41%
8192	0.26%	0.55%	1.14%	2.32%	4.70%	9.53%	19.49%
9216	0.23%	0.49%	1.01%	2.06%	4.17%	8.46%	17.25%
10240	0.21%	0.44%	0.91%	1.85%	3.75%	7.60%	15.47%
11264	0.19%	0.40%	0.83%	1.68%	3.41%	6.89%	14.02%
12288	0.17%	0.37%	0.76%	1.54%	3.12%	6.31%	12.82%
13312	0.16%	0.34%	0.70%	1.42%	2.88%	5.82%	11.81%
14336	0.15%	0.31%	0.65%	1.32%	2.67%	5.40%	10.95%
15360	0.14%	0.29%	0.61%	1.23%	2.49%	5.04%	10.20%
16384	0.13%	0.27%	0.57%	1.16%	2.34%	4.72%	9.55%
.	.	.	.	.	.	.	.
32768	0.06%	0.14%	0.28%	0.58%	1.17%	2.35%	4.73%

- the algorithms with the fission loop transformation and loop interchange, that is: `fission-row-ij` (Figure 3) and `fission-row-ji` (Figure 4) for the row-wise layout and `fission-col-ij`, `fission-col-ji` for the column-wise layout;

and two types of versions of WZ factorization block algorithms:

- the strip-mining algorithms: `basic-row-sm-`b, `basic-col-sm-`b;
- the loop tiling algorithms: `fission-row-ij-lt-`b, `basic-row-ji-lt-`b, `fission-col-ij-lt-`b, `fission-col-ji-lt-`b.

In the notation `sm` is short for strip-mining, `lt` for loop tiling, and b is the `BLOCK_SIZE`. The following block sizes are checked: 8, 16, 32, 64, 128, 256, 512.

All versions have been implemented in C++ with vectorization and parallelization.

For testing, we used a double-precision square matrix of random values. The size of the smallest matrix is R (rows times columns). All sizes are shown in Table II.

TABLE II: Characteristics of the test data sizes.

Data size	n	Number of cells ($n \times n$)	[GB]
R	8192	67108864	0.5
2.25R	12288	150994944	1.125
4R	16384	268435456	2
16R	32768	1073741824	8

The performance and energy consumption tests of the proposed versions of the WZ algorithm were carried out on the following computing platform equipped with a modern multicore processor with the following parameters and software:

- processor: Intel(R) Xeon(R) Gold 5218R (2.10 GHz; HT; 2×20 cores;
- operating system: CentOS 7.5 with Linux kernel 3.10.0;
- compiler: Intel ICC 14.0.2 with compiler options `-qoenmp -03 -ipo -no-prec-div -fp-model fast=2`

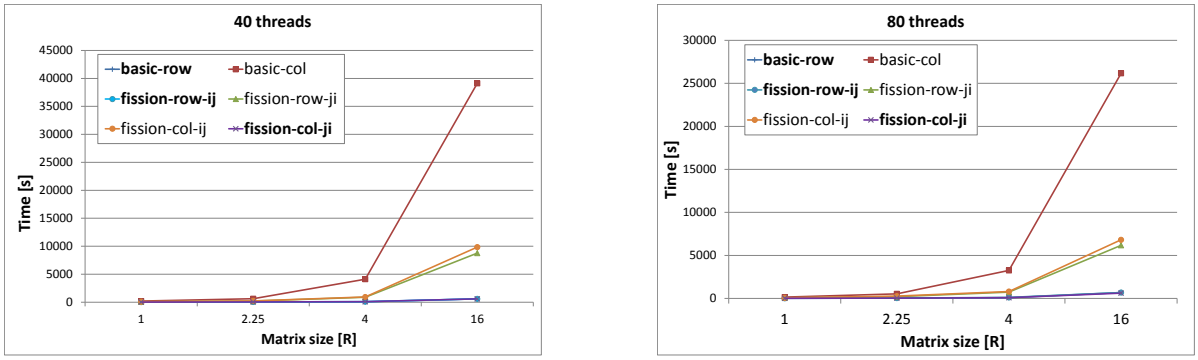
To analyze the impact of all versions of the algorithm on energy consumption, we used measurements from the RAPL (Running Average Power Limit) interface. We used RAPL because the article [9] has shown that it gives the correct measurement results.

B. Execution Time. Matrix layout and loop interchange

First, we measure the execution time of the following versions of WZ factorization algorithm: the basic algorithm and the loop fission algorithm for the matrix stored in memory in two formats: a row-wise and column-wise manner and loop interchanged the order of the i-loop and j-loop. Therefore, we test 6 versions of the algorithm:

- `basic-row`
- `basic-col`
- `fission-row-ij`
- `fission-row-ji`
- `fission-col-ij`
- `fission-col-ji`

We test all these versions on 40 threads as well as on 80 threads with running HT. The results are presented in Figure 10. Both graphs on the vertical axis have execution time and on the horizontal axis the data size. Also, both of them show lines with points indicating the time of the algorithm for different data sizes. The graph in Figure 10a shows the runtime of the algorithm running on 40 threads, whereas the graph in Figure 10b shows the runtime of 80 threads, respectively. In both graphs we see 6 lines, each of them concerns one version of the algorithm according to the legend. Also in both cases, the lines for the `basic-row`, `fission-row-ij`, and `fission-col-ji` algorithms almost overlap. As we can see from the graphs, the above-mentioned versions also have the shortest runtime. Table III shows the speedup we get



(a) 40 threads

(b) 80 threads

Fig. 10: Execution time of versions of WZ factorization algorithm for different data sizes.

on 80 threads compared to 40 threads for each version of the tested WZ algorithm, which is defined as follows:

$$S = \frac{T_{40th}}{T_{80th}}$$

where parameter T_{40th} denotes the execution time of the algorithm run on 40 threads and T_{80th} denotes the execution time of the algorithm run on 80 threads.

TABLE III: Relative speedup of versions of WZ algorithm operating on 40 and 80 threads (T_{40th}/T_{80th}).

	R	2.25R	4R	16R
basic-row	**0.91**	**0.95**	**0.88**	**0.85**
basic-col	1.21	1.15	1.26	1.49
fission-row-ij	**0.93**	**0.81**	**0.70**	**0.85**
fission-row-ji	1.11	1.03	1.21	1.41
fission-col-ij	1.22	0.88	1.15	1.44
fission-col-ji	**0.96**	**0.89**	**0.90**	**0.96**

Based on Table III, we can observe that, regardless of the size of data, the acceleration of the operating time between the operation on 40 threads and on 80 threads is of a similar order. In Table III, the bold lines refer to the versions, which, as we could see in the graphs in Figure 10, fared better in terms of runtime, i.e. `basic-row`, `fission-row-ij`, and `fission-col-ji`. We can see for them that running them on 80 threads causes an increase in the runtime (values below 1), contrary to expectations. However, we can also notice that some algorithms speed up when they are run on 80 threads (values above 1) but this applies to versions which, as we could see from the graphs in Figure 10, had a longer runtime. For those versions of the algorithm that perform better in terms of runtime, the machine parameters are sufficient, therefore running HT for them does not improve their runtime. On the other hand, for those that perform weaker in terms of runtime, the capabilities of the machine are not sufficient, therefore HT improves the runtime.

Although 80 threads give these versions some speedup, they still perform worse in runtime than `basic-row`, `fission-row-ij` or `fission-col-ji`. We can see that the versions with the shortest runtimes, i.e. `basic-row`, `fission-row-ij`, and `fission-col-ji`, perform better in terms of time on 40 threads. Therefore, we will conduct further tests only for the algorithm operating on 40 threads. We will carry out further considerations by selecting one version of the basic algorithm and one version of the fission algorithm. In the case of the basic version of the algorithm, we will choose the one for which we had a better runtime, i.e. `basic-row`.

However, in the case of fission versions, we have two versions, the runtime of which is better than the others and comparable to `fission-row-ij` and `fission-col-ji`, that is, those where the loop order was consistent with the matrix layout, therefore it is not surprising that they perform similarly. For further considerations, we will decide on any one of them, e.g. `fission-row-ij`, with the same matrix representation — row-wise — as for the chosen basic version.

Of the two algorithm versions selected for further tests, `basic-row` tends to perform slightly better than `fission-row-ij` in terms of runtime for different data sizes (at up to 12%).

C. Energy Consumption. Matrix layout and loop interchange

The graph from Figure 11 shows the energy consumption in joules for the `basic-row` (red) and `fission-row-ij` (blue) versions for all four data sizes.

We can see that less energy is consumed by the `basic-row` algorithm, we see this for each data size but it is a small amount between 1% and 11% percent. (respectively for data sizes: 10.33%, 1.33%, 10.98%, 2.63%). We can also observe that as the data size increases, the energy consumption also varies proportionally, and it is the following increase:

$$c_e(k) = k^{\frac{3}{2}}$$

where k is the data size growth factor in our case equal respectively: 1, 2.25, 4 and 16.

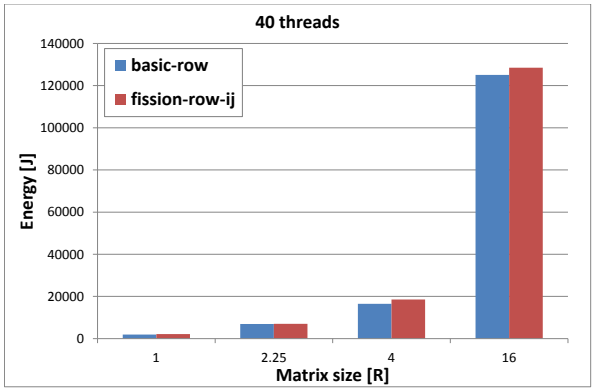


Fig. 11: Energy consumption for `basic-row` and `fission-row-ij` algorithms for different data sizes.

In general, energy consumption is to be expected proportional to the number of operations performed, and this is proportional to $k^{\frac{3}{2}}$.

TABLE IV: Energy consumption ratios as the data size increases.

	R	2.25R	4R	16R
basic-row	1	3.59	8.54	64.72
fission-row-ij	1	3.26	8.6	59.6
$c_e(k)$	1	3.37	8	64

Table IV shows how many times the energy consumption of individual versions has increased for each data size in relation to the energy consumption for the data size R.

D. Execution Time. Strip-mining and loop tiling

Now we will consider the block version of WZ factorization with strip-mining and loop tiling. We will only consider the algorithm implementations that gave the best results in the tests from the previous sections, limiting ourselves to two of them: `basic-row` and `fission-row-ij`. The effect of different `BLOCK_SIZE` on the efficiency will be tested experimentally. We will receive the following versions of the algorithm for further testing:

- `basic-row-sm-b`
- `fission-row-ij-lt-b`

The following block sizes (b) will be tested: 8, 16, 32, 64, 128, 256, 512. In total, 14 different versions of the algorithm will be tested. Our goal is to experimentally investigate which block size will work best in terms of the algorithm's runtime.

In Figure 12 we have graphs showing the runtime of the `basic-row-sm-b` versions for different block sizes b. Each graph deals with the operation of the algorithm on a different size of data. We can see that for individual data sizes (that is: R, 2.25R, 4R, 16R), the block algorithm is the fastest for the block sizes: 64, 64, 256 and 256, respectively, and the slowest for the block sizes: 8, 16, 8 and 8, respectively. The summary of these observations is presented in Table V.

In Table V the columns present information about the algorithm operating on the specified data size, that is, R, 2.25R, 4R, and 16R, respectively. The last line shows the percentage profit between the slowest and the fastest version of the algorithm, i.e. the profit resulting from the selection of the best-performing block size for the `basic-row-sm-b` versions of the algorithm and the specified data size.

Summarizing the data collected in Table V, we can see that we cannot clearly indicate one block size that would give equally good results for all data sizes. One can notice that for smaller data sizes: (R, 2.25R), the smaller block works better, while for larger data sizes (4R, 16R), the larger block works better. However, we can clearly see which block size perform the worst, these are smaller block size. We can then conclude that small blocks work poorly.

TABLE V: The best and the worst block size due to the `basic-row-sm-b` versions runtime for different data sizes.

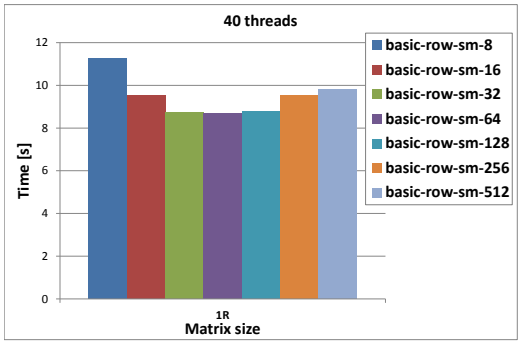
	R	2.25R	4R	16R
min. time [s]	8.67	28.90	68.54	556.17
The best block size	64	64	256	256
max. time [s]	11.26	32.53	85.32	601.59
The worst block size	8	16	8	8
max−mix [s]	2.58	3.63	16.78	45.42
%	23%	11%	20%	8%

Let's see what the situation looks like for the `fission-row-ij-lt-b` versions of the algorithm. In Figure 13 we have graphs showing the runtime of the `fission-row-ij-lt-b` versions for different block sizes. Here each graph deals with the operation of the algorithm on a different size of data too. We can see that for individual data sizes the algorithm is the fastest for the block sizes: 8, 8, 64, and 64, respectively, and the slowest for the block sizes: 512, 512, 8, and 8, respectively. The summary of these observations is presented in Table VI.

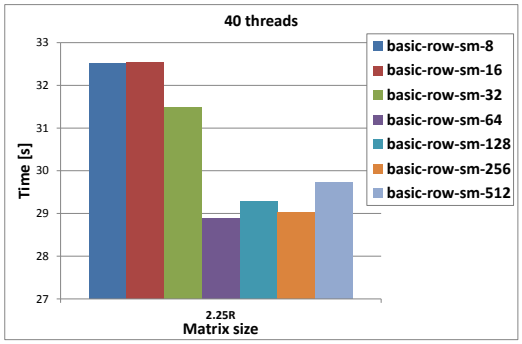
TABLE VI: The best and the worst block size due to the `fission-row-ij-lt-b` versions runtime for different data sizes.

	R	2.25R	4R	16R
min. time [s]	11.00	34.03	89.69	681.83
The best block size	8	8	64	64
max. time [s]	18.77	48.78	198.51	1904.31
The worst block size	512	512	8	8
max−mix [s]	7.77	14.75	108.82	1222.48
%	41%	30%	55%	64%

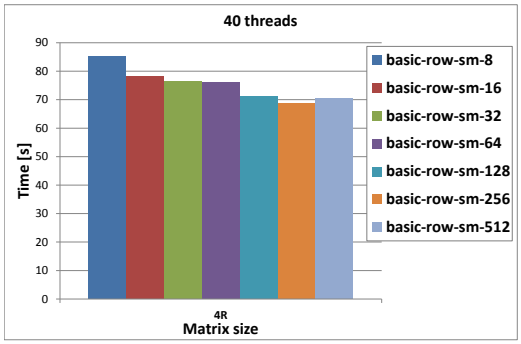
Looking at Figures 12, 13 and Tables V, VI, we can see that the `basic-row-sm-b` versions of the algorithm is better in terms of runtime than the `fission-row-ij-lt-b` versions, regardless of the selection of the block size. In the case of the `fission-row-ij-lt-b` versions of the algorithm, we can no longer conclude that, in general, regardless of the size of the data, small blocks perform worse. However, we



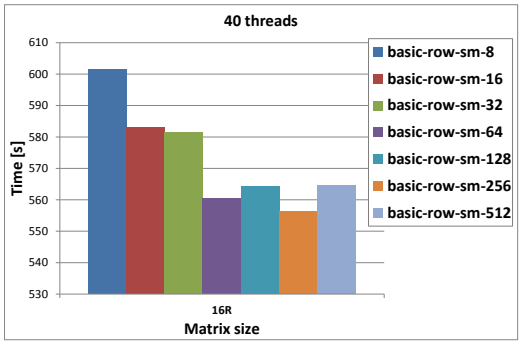
(a) Data size R



(b) Data size 2.25R



(c) Data size 4R



(d) Data size 16R

Fig. 12: Execution time of `basic-row-sm-b` versions for different data sizes and different block sizes.

can still observe that as the data size increases, the block size that works best also grows.

From the tables above, we can see that choosing the correct block size can save up to 23% of the time of the `basic-row-sm-b` versions of the algorithm (Table V), and even 64% in the case of the `fission-row-ij-lt-b` versions of the algorithm (Table VI).

E. Energy Consumption. Strip-mining and loop tiling.

Tables VII and VIII provide a summary of the energy consumption during the operation of the various versions of the tested algorithms. The last line shows the percentage energetic profit between the most and the least energy-consuming version of the algorithm, i.e. the profit resulting from the selection of the best-performing block size for the algorithm and the specified data size. Accordingly, Table VII presents data for the `basic-row-sm-b` versions of the algorithm and Table VIII for the `fission-row-ij-lt-b` versions.

We can see that choosing the right block is very important because it can save us 22% of energy consumption in the case of the `basic-row-sm-b` version of the algorithms (Table VII) and up to 61% of energy consumption in the case of the `fission-row-ij-lt-b` versions of the algorithm (Table VIII).

TABLE VII: The best and the worst block size due to energy consumption of the `basic-row-sm-b` algorithm for different data sizes.

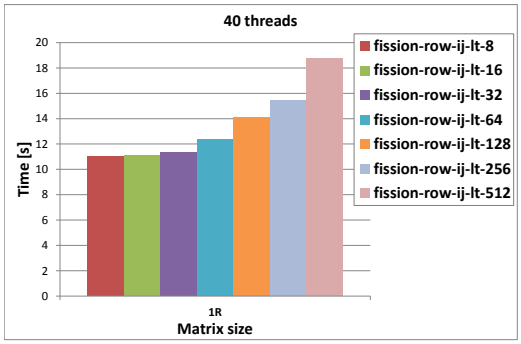
	R	2.25R	4R	16R
min [J]	1687.53	6152.07	14925.4	118442.00
The best block size	64	64	256	256
max [J]	2159.86	6827.51	18542.5	129889.00
The worst block size	8	16	8	8
max−mix [J]	472.32	675.44	3617.03	11447.00
%	22%	10%	20%	9%

TABLE VIII: The best and the worst block size due to energy consumption of the `fission-row-ij-lt-b` algorithm for different data sizes.

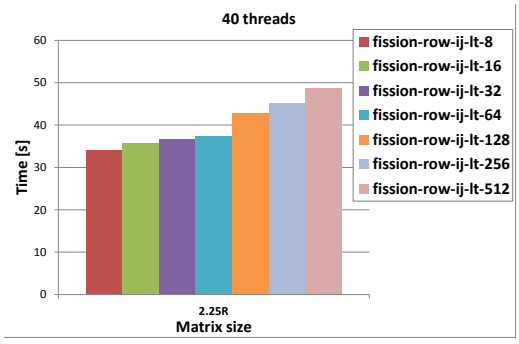
	R	2.25R	4R	16R
min [J]	2189.55	7180.41	18640.1	143575.00
The best block size	32	8	64	64
max [J]	3863.04	10154.80	39273.60	370432.00
The worst block size	512	512	8	8
max-mix [J]	1673.49	2974.36	20633.60	226858.00
%	43%	29%	53%	61%

F. Time execution-energy trade-off

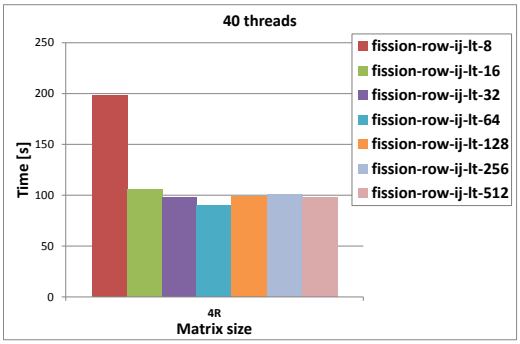
Although this is usually the case, the best runtime does not always result in the best energy consumption. We can see it



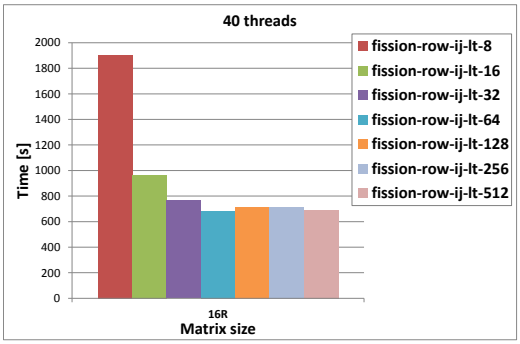
(a) Data size R



(b) Data size 2.25R



(c) Data size 4R



(d) Data size 16R

Fig. 13: Execution time of `fission-row-ij-lt-b` versions for different data sizes and different block sizes.

in Tables VI and VIII which show data from experiments on block versions of the fission algorithm, where for the size R, data block size giving the best result in terms of runtime is 8, while the best in terms of consumption energy proved to be 32.

Finally, we present in Table IX a summary of energy consumption, performance, and energy efficiency of the algorithm versions, which are the best during the experiences, for the largest tested data size (16R).

Analyzing the data in Table IX, we can see that `basic-row-sm-256` is the best among the best versions of the algorithm. In terms of energy efficiency, it is 6% better than the second in line, i.e. `basic-row`. However, for smaller data sizes, it turned out to be the best version `basic-row-sm-64` (see Table V and Table VII).

V. Conclusion

This article investigates four loop transformation strategies for the WZ factorization, namely: loop fission, loop interchange (permutation), strip-mining and loop tiling. The loop transformation affects both runtime and energy consumption. It can have both a positive effect in reducing runtime and energy consumption and a negative effect in increasing runtime and energy consumption. Measurements were made on a 2nd Generation Intel Xeon Scalable Processors using the Intel RAPL interface.

Our experiments have shown that the `basic-row` version is definitely better in terms of runtime than the `basic-col`. The advantage of the former is the greater, the larger the size of the data we process and our tests show that it ranges from 19 to even 60 times faster, it is `basic-row`, we can see it in the graph in Figure 10. The first described experiments also showed that HT does not bring benefits in our case.

Our tests have also shown that the loop interchange transformation we propose has a large impact on the reduction of calculation time and energy consumption. The versions for which the loop interchange was compatible with the matrix representation, i.e. `fission-row-ij` and `fission-col-ji`, perform better in terms of operating time. They fell out the better, the larger the size of the data was processed and our experiments showed that it was from 6 to 16 times faster than in the case of a loop interchange inconsistent with the matrix representation. So we should never choose a loop interchange inconsistent with the matrix representation.

However, when comparing the energy consumption for the `basic-row` and fission versions with the loop interchange

TABLE IX: Energy efficiency for four best versions of the algorithm (dataset: 16R).

Versions	Time [s]	Total energy [J]	Performance [Gflops/s]	Energy efficiency[Gflops/J]
basic-row	582.63	125098.72	40.26	0.19
fission-row-ij	588.15	128481.50	39.88	0.18
basic-row-sm-256	**556.17**	**118442.28**	**42.17**	**0.20**
fission-row-ij-lt-64	681.83	143574.55	34.40	0.16

consistent with the matrix representation `fission-row-ij`, we saw that the `basic-row` version was slightly better from 1% to 11% less energy consumption (Figure 11). So the fission transformation won't pay off.

Finally, our experiments have shown that the best version among block versions depends on the data size, and here block size must be selected experimentally. The only thing we can see is that as the data size increases, the block size also increases. It may turn out that if the block size is poorly selected, the energy consumption may be higher by up to 61%. For 16R data size, they are versions `basic-row-sm-256` and `fission-row-ij-lt-64` which works the best (see Table IX). Moreover, we can say that regardless of the size of the data, the application of the strip-mining transformation worked best. For data size 16R the `basic-row-sm-256` version turned out to be the most profitable, as can be seen in Table IX. On the other hand, the transformation of loop tiling does not pay off because it causes a lot of complications in the code and it gives a slight extension of the runtime and slightly higher energy consumption.

Future work includes extending our experimental comparison to a wide range of architectures, including graphics cards. In addition, we will evaluate the performance impact of various runtime systems for OpenMP configurations and loop transformation energy for the WZ and the three decomposition main kernels in dense linear algebra algorithms (Cholesky, LU, and QR).

REFERENCES

[1] Beata Bylina and Jarosław Bylina. OpenMP Thread Affinity for Matrix Factorizationon Multicore Systems. *Proceedings of the Federated Conference onComputer Science and Information Systems*, 11:489–492, 2017. https://doi.org/10.15439/2017F231.

[2] Beata Bylina and Jarosław Bylina. Nested loop transformations on multi- and many-core computers with shared memory. In *Selected Topics in Applied Computer Science*, volume I, pages 167–186. Maria Curie-Skłodowska University Press, Lublin, 2021. http://stacs.matrix.umcs.pl/v01/stacs_v01.pdf.

[3] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computation: Practice and Experience*, 27(5):1292–1309, 2014. https://doi.org/10.1002/cpe.3306.

[4] D.J. Evans and M. Hatzopoulos. A parallel linear system solver. *International Journal of Computer Mathematics*, 7(3):227–238, 1979. https://doi.org/10.1080/00207167908803174.

[5] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. *SIGPLAN Not.*, 40(6):315–326, June 2005. https://doi.org/10.1145/1064978.1065048.

[6] Fred G. Gustavson. *New Generalized Matrix Data Structures Lead to a Variety of High-Performance Algorithms*, pages 211–234. Springer US, Boston, MA, 2001. https://doi.org/10.1007/978-0-387-35407-1_13.

[7] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the Intel Haswell processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904, 2015. https://doi.org/10.1109/IPDPSW.2015.70.

[8] Vasilios Kelefouras and Karim Djemame. A methodology for efficient code optimizations and memory management. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, page 105–112, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3203217.3203274.

[9] K. Khan, M. Hirki, T. Niemi, J. Nurminen, and Z. Ou. RAPL in action: Experiences in using RAPL for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3, 01 2018. https://doi.org/10.1145/3177754.

[10] Martin Kong and Louis-Noël Pouchet. Model-driven transformations for multi- and many-core CPUs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 469–484, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3314221.3314653.

[11] João V.F. Lima, Issam Raïs, Laurent Lefevre, and Thierry Gautier. Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 7–12, 2017. https://doi.org/10.1109/SBAC-PADW.2017.10.

[12] João Vicente Ferreira Lima, Issam Raïs, Laurent Lefèvre, and Thierry Gautier. Performance and energy analysis of OpenMP runtime systems with dense linear algebra algorithms. *The International Journal of High Performance Computing Applications*, 33(3):431–443, 2019. https://doi.org/10.1177/1094342018792079.

[13] Maxime Mirka, Guillaume Devic, Florent Bruguier, Gilles Sassatelli, and Abdoulaye Gamatié. Automatic energy-efficiency monitoring of openmp workloads. In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 43–50, 2019. https://doi.org/10.1109/ReCoSoC48741.2019.9034988.

[14] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. *SIGPLAN Not.*, 46(1):549–562, January 2011. https://doi.org/10.1145/1925844.1926449.

[15] Yukinori Sato, Tomoya Yuki, and Toshio Endo. An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation. *ACM Trans. Archit. Code Optim.*, 15(4), January 2019. https://doi.org/10.1145/3293449.

[16] Md Abdullah Shahneous Bari, Abid M. Malik, Ahmad Qawasmeh, and Barbara Chapman. Performance and energy impact of openmp runtime configurations on power constrained systems. *Sustainable Computing: Informatics and Systems*, 23:1–12, 2019. https://doi.org/10.1016/j.suscom.2019.04.002.

[17] Przemysław Stpiczyński. Vectorized algorithm for multidimensional Monte Carlo integration on modern GPU, CPU and MIC architectures. *J. Supercomput.*, 74(2):936–952, February 2018. https://doi.org/10.1007/s11227-017-2172-x.

[18] Aleksandar Vitorović, Milo V. Tomašević, and Veljko M. Milutinović. Chapter five - manual parallelization versus state-of-the-art parallelization techniques: The spec cpu2006 as a case study. In Ali Hurson, editor, *Advances in Computers*, volume 92 of *Advances in Computers*, pages 203 – 251. Elsevier, 2014. https://doi.org/10.1016/B978-0-12-420232-0.00005-2.

[19] P. Yalamov and D.J. Evans. The WZ matrix factorisation method. *Parallel Computing*, 21(7):1111–1120, 1995. https://doi.org/10.1016/0167-8191(94)00088-R.