Original published: T. Jakobs, M. Hofmann, and G. Rünger. Reducing the power consumption of matrix multiplications by vectorization. In *Proceedings of the 19th IEEE International Conference on Computational Science and Engineering (CSE 2016)*, pages 1–8. IEEE, August 2016. Online available at http://dx.doi.org/10.1109/CSE-EUC-DCABES.2016.187.

Reducing the Power Consumption of Matrix Multiplications by Vectorization

Thomas Jakobs, Michael Hofmann, Gudula Rünger
Technische Universität Chemnitz
Department of Computer Science
Professorship of Practical Computer Science
E-mail: [thomas.jakobs, mhofma, ruenger]@informatik.tu-chemnitz.de

***Abstract*—The power consumption of programs and algorithms is currently a very active research field. This includes the investigation of the effect of different programming techniques on power consumption. Some programming techniques have already been studied intensively. However, there are techniques that did not get as much attention as needed so far. One of these techniques is the vectorization of programs, which uses special operations to calculate several data in one step. In this article, we investigate the effect of vectorization on the power consumption and study several program versions of dense matrix multiplication which combine vectorization with other techniques, such as loop unrolling or compiler options. We show that the use of vectorization is not only capable of improving the performance but can also reduce the power and energy consumption of programs.**

I. Introduction

The power consumption of modern computer systems gains more and more importance and is a main concern especially in computing centers where the amount of energy transferred to the center is a limiting factor in computations and cooling. Considerations of energy and power savings are already made during the design phase of the hardware system. However, the behavior of software with respect to power and energy is equally important. To study the energy and power behavior of software, a general knowledge of the behavior of different programming and design techniques is required, which has been an active field of research during the last years. A programming technique that did not get much attention so far in recent research is the vectorization of programs.

The use of vectorized algorithms is becoming increasingly important in current computer systems. This is due to the fact that for most processors the speed limit of the processor has reached its maximum. To bypass this limit, multiprocessor systems were invented that can run different threads in parallel on a set of cores. This leads to the performance being limited by the number of cores and the suitability of the program to be parallelized. Another approach to execute programs faster is the use of the built-in SIMD operations with which several data can be synchronously calculated in one step. The modification of a program such that it uses such SIMD operations is called vectorization. The principles of vectorization are well known from vector computers and more recently since the introduction of the Pentium Processors with *Multi Media Extension* (MMX) and its successor *Streaming SIMD Extensions* (SSE). Today, the *Advanced Vector Extensions* (AVX) support vectors that can hold up to 8 single precision floating point values which can be processed in one step. This makes it possible to speed up a synchronous calculation and leaves the possibilities of multi-threading for asynchronous calculations.

The programming of SIMD units can either be done with sequential code that is auto-vectorized by the compiler or it can be programmed using compiler specific intrinsic functions. The differences in performance achieved by the two approaches are already known from other fields of research, see [1], [2]. In our work, among other things, we investigate whether these differences also apply to the power consumption of an algorithm.

The investigations are done for two Intel® processors, of the Sandy Bridge and the Haswell architecture, and the SIMD instructions AVX. As a typical example from synchronous linear algebra algorithms, we have chosen the matrix multiplication for dense matrices, which is a kernel in many scientific simulation codes. Starting with a first basic sequential algorithm, we provide two vectorized versions, which are an auto-vectorized program version and a manually vectorized program version using AVX intrinsics. In addition, we apply loop unrolling in three different ways resulting in three program versions for each basic version. Moreover, different compiler optimization levels are used and environmental features, such as processor frequency, are changed. All these program versions have been studied concerning their power and energy behavior. The goal is to find out whether one or more of the presented program code modifications reduce the consumption of energy and/or power during execution time.

The rest of this article is structured as follows. Section II introduces the initial problem for matrix multiplication and several program versions. Measurement results of energy and power consumption for the program versions are presented and discussed in Section III. A brief overview over related research is given in Section IV and Section V concludes the article.

```
for(i < l; stride 1) {
  for(j < n; stride 8) {
    C[i][j] = 0
    ...
    C[i][j+7] = 0
    for(k < m; stride 1) {
      C[i][j] += A[i][k] * B[k][j]
      ...
      C[i][j+7] += A[i][k] * B[k][j+7]
    }
  }
}
```

Listing 1: Basic algorithm for calculating a matrix multiplication with unrolled loop body of 8 elements (named Seq8 in the following).

```
for(i < l; stride 1) {
  for(j < n; stride 8) {
    c0 = _mm256_setzero_ps()
    for(k < m; stride 1) {
      a0 = _mm256_broadcast_ss(A[i][k])
      b0 = _mm256_load_ps(B[k][j])
      t = _mm256_mul_ps(a0, b0)
      c0 = _mm256_add_ps(t, c0)
    }
    _mm256_store_ps(C[i][j], c0)
  }
}
```

Listing 2: Vectorized algorithm of the matrix multiplication given in Listing 1 using intrinsic functions given in Table 1 (named In8 in the following).

II. Problem statement

In this article, we investigate whether the energy or power consumption of an algorithm can be reduced by using vectorization techniques. For our investigations, we use an algorithm for a dense matrix multiplication given by the following equation:

$$C[i,j] = \sum_{k=1}^{m} A[i,k] \cdot B[k,j]$$
$$\text{where } i = 1 \dots l; j = 1 \dots n$$
$$\text{and } l, m, n \in \mathbb{N}$$

A. Program versions

Matrix multiplications are commonly used subroutines in many algorithms of scientific computing, which makes its studies widely interesting. We use the well known optimization techniques loop unrolling in combination with loop tiling to create different versions of the matrix multiplication. By loop unrolling the body of a loop is packed several times into one new loop body within a loop with adjusted loop counters. The advantage of loop unrolling is that the effects of pipelining and caching can be utilized in a more sophisticated way by the compiler. A basic matrix multiplication algorithm with an unrolled loop body is shown in Listing 1.

Intrinsic	Description
_mm256_setzero_ps	set elements of vector to 0
_mm256_broadcast_ss	load one element and copy it to all vector elements
_mm256_load_ps	load 8 consecutively stored elements into vector
_mm256_mul_ps	multiply two vectors
_mm256_add_ps	add two vectors
_mm256_fmadd_ps	multiply two vectors then add a third
_mm256_store_ps	store vector elements to 8 consecutive memory positions

Table 1: Overview of AVX intrinsic functions used and their corresponding purpose.

Algorithm	Sequential	Auto-vectorized	Intrinsic
1x8 loop unrolling	Seq8	Av8	In8
3x24 loop unrolling	Seq24	Av24	In24

Table 2: The table shows the overview of the program versions implemented. It also gives the names of the implementations used in this article.

We compare a sequential, an auto-vectorized and an intrinsic program version of the matrix multiplication. To enforce the sequential (shown in Listing 1) and the auto-vectorized program versions to be compiled entirely without or with auto-vectorization respectively, we have used the `#pragma` directives of the Intel® Compiler [3]. These `#pragma` directives are `novector` for the sequential version and `vector always` for the auto-vectorized version.

The intrinsic code has been implemented using the Intel® Compiler intrinsics as shown in Listing 2. Intrinsics represent special assembler commands in a function-wrapped form. The specific intrinsic functions used in this article are given in Table 1. These intrinsic functions begin with a specifier for their domain (`mm256` for AVX) followed by a specific name for a function and the data types used (e.g. `ps` for `float`), see [3].

Table 2 gives the various program versions which result from the different types of loop unrolling and introduces their respective names used in this article. The loop unrolling stages are shown in short names (e.g. 1x8), where the first number represents the unrolling factor of the outermost loop (`i < l` in Listing 1) and the second factor represents the second loop (`j < n` in Listing 1). These combinations are each implemented as separately coded algorithms.

As an example for the combination of vectorization and loop unrolling, Listing 3 shows the 3x24 unrolled algorithm using intrinsic functions. Additionally, Figure 1 gives an overview on how the use of loop unrolling creates a blockwise pattern in the innermost loop. This program version gives the ability to use loaded data for more than one calculation of an intermediate result c_i. In detail, one load operation can be used for the calculation of three intermediate result values c_i, which shifts the ratio of loaded vectors to calculated vectors from 2:1 in the basic program version of Listing 1 to a ratio of 2:3 (6 loads:9 results) in the 3x24

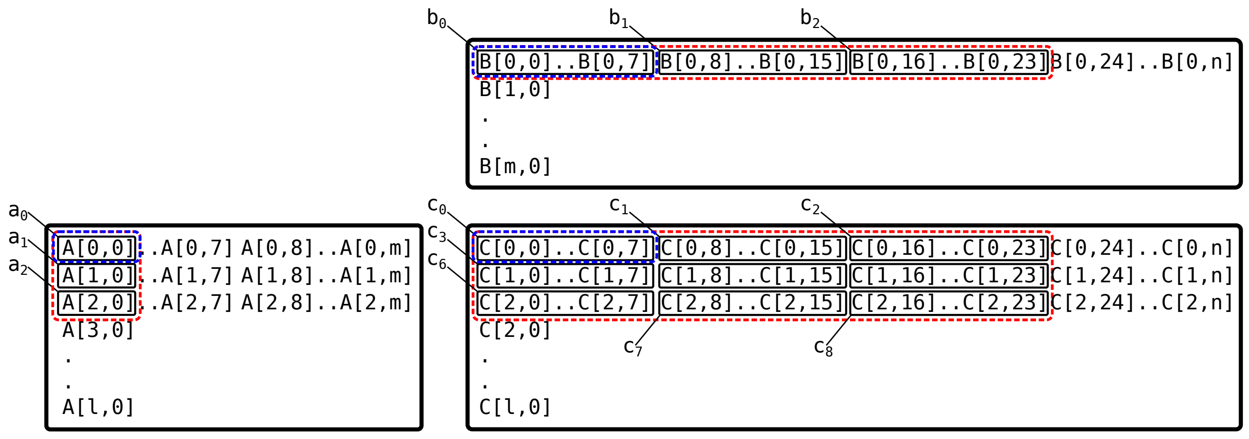

Figure 1: Schematic view of a dense matrix multiplication with the use of AVX-vectors (black boxes) containing 8 `float` values. The single values of matrix A are copied (`broadcast`) into each of the vector elements. The blue dotted lines show the operands of the innermost loop without loop unrolling. The red dotted lines highlight the operands of the innermost loop in the 3x24 unrolled program version. This loop unrolling leads to a block-wise calculation of intermediate result values c_i in matrix C.

```
for(i < l; stride 3) {
  for(j < n; stride 24) {
    c0 = _mm256_setzero_ps()
    ...
    c8 = _mm256_setzero_ps()
    for(k < m; stride 1) {
      a0 = _mm256_broadcast_ss(A[i][k])
      a1 = _mm256_broadcast_ss(A[i+1][k])
      a2 = _mm256_broadcast_ss(A[i+2][k])
      b0 = _mm256_load_ps(B[k][j])
      b1 = _mm256_load_ps(B[k][j+8])
      b2 = _mm256_load_ps(B[k][j+16])
      t  = _mm256_mul_ps(a0, b0)
      c0 = _mm256_add_ps(t, c0)
      ...
      t  = _mm256_mul_ps(a2, b2)
      c8 = _mm256_add_ps(t, c8)
    }
    _mm256_store_ps(C[i][j], c0)
    ...
    _mm256_store_ps(C[i+2][j+16], c8)
  }
}
```

Listing 3: Vectorized algorithm using intrinsics for block-wise calculation of a matrix multiplication (named In24 in the following). The outer loop is unrolled 3 times and the second loop is unrolled 24 times.

unrolled program version. Since calculation operations are the desired effect of the program, whereas load and store operations take up uncredited time, this shift increases the utilization of used hardware for the matrix multiplication. Since the 3x24 unrolling program version already uses all of the 16 available AVX-registers a further unrolling is not reasonable.

In addition to vectorization, we are further interested in the influence of compiler optimizations, of different frequencies and of different hardware architectures on the power and energy consumption.

B. *Execution environment and software*

The experiments have been executed on a Linux Debian utilizing kernel version 3.16.0 and additional modules for the usage of Model Specific Registers (MSRs) and frequency manipulation. In order to measure the pure effects of the energy consumption of the matrix multiplication the desktop manager has been disabled and all unnecessary processes have been turned off. The Intel® Compiler `icc` with additional flags `-std=c++11 -restrict -xavx` has been used.

Program code that is compiled can be optimized by the compiler in different ways. When implementing an algorithm with untypical technology, such as vectorization, the compiler might not be able to apply all optimizations that can be applied to standard code. This may lead to a difference in energy and power consumption of the program version resulting from a different set of compiler optimizations used. The interesting point is whether the power consumption of a vectorized algorithm can be further reduced by the use of a special optimization level. These optimization levels are called from O0 (no optimization) to O3 (aggressive optimization) according to the command line arguments handed over to the compiler.

The execution of the implementations with different frequencies is done by setting the CPU clock speed to a fixed value. This can be done using so called governors provided by the operating system and the frequency manipulation module. One of the used governors is the `userspace` governor that allows setting a fixed processor frequency out of a predefined set of values. Additionally, dynamic frequency governors can alter the frequency during the runtime of a program. There are several governors available, each trying to achieve a specific goal with its frequency modifications. One example is the `powersave` governor

Processor	Sandy Bridge	Haswell
Level 3 Cache	8 MB	8 MB
# of Cores	4	4
# of Threads	8	8
Base Frequency	3,4 GHz	3,5 GHz
Turbo Frequency	3,8 GHz	3,9 GHz
Memory Bandwidth	21 GB/s	25,6 GB/s

Table 3: Brief specification overview of processors used.

frequency	varying	max	dynamic
variable matrix sizes		✓	
fixed matrix size of 3360	✓	✓	✓
Optimization Levels (O0 - O3)			✓

Table 4: Overview of combinations for all implementations of Table 2 used for measuring.

which uses the frequency of the processor to execute the programs with a low power consumption.

C. Utilized hardware

For the execution of the given implementations, we have used two different Intel® architectures, which are an i7-2600 processor of the Sandy Bridge architecture and an i7-4770K of the Haswell architecture. We have chosen these two processors, since they have different architectures but are similar in their architectural properties which are given in Table 3.

The different architectures of the two processors also lead to different AVX intrinsics available. More precisely, the extension AVX2 is additionally available on the Haswell processor. Using this extension, it is possible to use *fused-multiply-add* (FMA) instructions, with which a multiplication and an addition can be done in one machine operation rather than first multiplying, storing the intermediate result and then adding another vector.

III. Experiment/Measurement

Intel® *Running Average Power Limits* (RAPL) is a well known interface for measuring the energy and power consumption of an Intel® CPU on software level [4]. It is used in many applications especially to determine the impact of software changes to the overall power consumption of a computer system. Using the RAPL interface makes it possible to read the values of energy consumption for the computation units or for the whole CPU.

We have measured the energy consumption of the different versions of a matrix multiplication under different conditions. Each of the combinations has been run and measured ten times and the mean value has been calculated and is shown in the graphs. Additionally, the sample variance is calculated and shown as error bar in the graphs.

The Intel® RAPL Interface is used to measure the energy and the time consumption of the different implementations. These energy values are then used to calculate the power consumption values using the formula $Power = \frac{Energy}{Time}$.

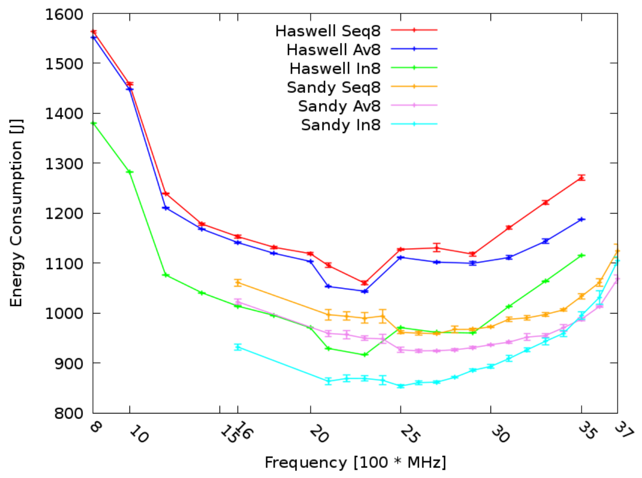

Figure 2: Energy consumption of basic (8 times unrolled) algorithms sequential, auto-vectorized and intrinsic for both architectures depending on CPU frequency.

In the following, we show the power and energy consumption with respect to different aspects, which are frequency and architecture, auto-vectorization, loop unrolling, matrix sizes, and compiler optimization levels. An overview of the program versions that have been run is given in Table 4. We first consider the energy consumption depending on the frequency.

A. Effects of vectorization on energy consumption

Figure 2 shows the energy consumption of the implementations given in Table 2 on both architectures. First, it can be observed, that the execution on the Sandy Bridge architecture uses less energy than on the Haswell architecture. On each architecture, the auto-vectorized algorithm consumes up to 7% less energy than the sequential algorithm. The intrinsic algorithm consumes even less energy which adds up to an energy saving of up to 15% compared to the sequential implementation.

Figure 3 shows the energy consumption values for the block-wise algorithms Seq24, Av24 and In24, which are lower for all algorithms than the energy values in Figure 2. The relation of the energy values of the different implementations remains the same as for the basic algorithm. However, the differences are higher and the energy saving of the auto-vectorized implementations can be up to 26% and the energy saving of the intrinsic algorithm can be up to 65% compared to the Seq24 program version. This shows that the consideration of optimization techniques, such as loop unrolling and intrinsics, can save vast amounts of energy. Additionally, the results show that the highest energy saving can be achieved by using a medium value for the frequency between the minimum and maximum frequency. The optimum is reached for a frequency of 2.3 GHz for the Haswell architecture and 2.5 GHz for the Sandy Bridge architecture.

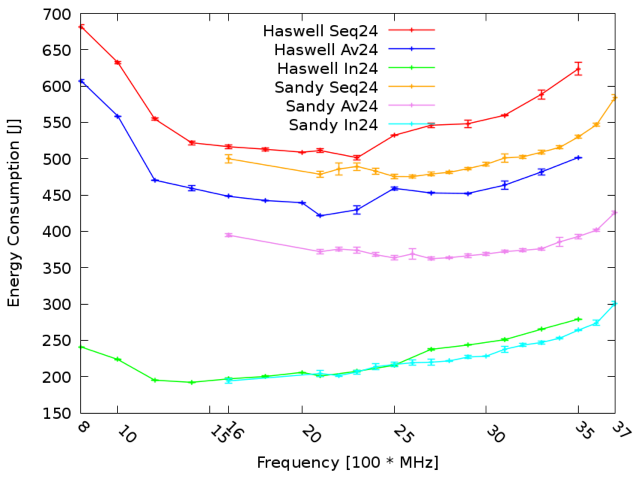

Figure 3: Energy consumption of block-wise (3x24 times unrolled) algorithms sequential, auto-vectorized and intrinsic for both architectures depending on CPU frequency.

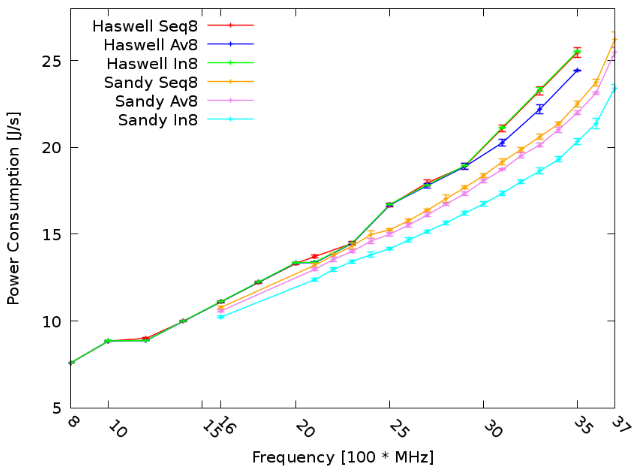

Figure 4: Power consumption of basic (8 times unrolled) algorithms sequential, auto-vectorized and intrinsic for both architectures depending on CPU frequency.

The important question to ask here is: Is the energy reduction only due to a shorter runtime or are there effects taking place that additionally save energy? This question will be answered in the following sections by investigating the power consumption of the matrix multiplication with respect to different influences.

B. Correlation of power consumption and frequency

In [5] and [6], it has already been shown that the power consumption of a sequential algorithm can be significantly reduced when it is executed with a lower CPU frequency. These findings also apply to the use of vectorization techniques. This is shown in Figures 4 and 5 in which the sequential program versions show the same qualitative behavior as the vectorized versions. The measurements also show that the power consumption follows a convex

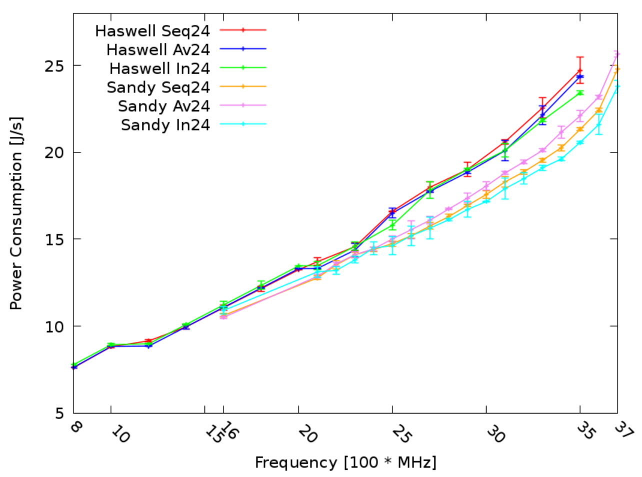

Figure 5: Power consumption of block-wise (3x24 times unrolled) algorithms sequential, auto-vectorized and intrinsic for both architectures depending on CPU frequency.

curve, leading to a higher power consumption for higher frequencies. The overall measurements show that the power consumption is about 70% lower for the execution with the lowest frequency compared to the highest possible frequency. Additional measurements show that the number of Level 1 Cache Misses is nearly constant over the range of all frequencies. Thus, the power saving of the algorithm with lower frequencies is not resulting from a different number of memory waits.

As the measurements shown in Figures 4 and 5 indicate, the execution on a Haswell architecture in general consumes more power. More precisely, the Sandy Bridge architecture consumes about 10% less power when running on the same frequency. On second glance, it is visible that the maximum power both architectures consume at their maximum frequency is nearly the same. This might follow from a design decision possibly made for the Haswell architecture to limit the frequency to a certain amount of power consumption.

With the launching of the Haswell architecture, also a new subset of AVX commands became available. One of the important new features are the FMA instructions that process one addition and one multiplication in one step. We have compared the FMA instruction against the original AVX command-set, i.e. combining the `add` and `mul` operation to one `fmadd` instruction. When comparing the results, there are different outcomes for the basic and the block-wise implementation.

For the basic implementation In8, there is no difference when using FMA commands as shown in Figure 6. When taking the energy consumption into account, it turns out to be the same values, too. This can be due to the executed code waiting for memory transfers, which in turn is limiting the execution speed.

For the block-wise implementation In24 the measurements in Figure 6 show that the use of FMA instructions

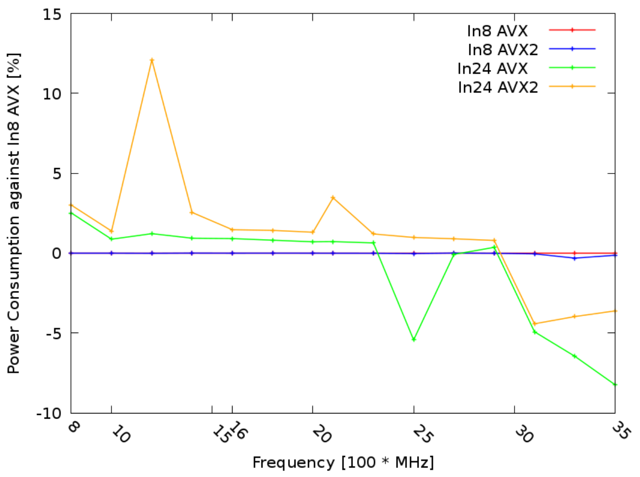

Figure 6: Power consumption intrinsic algorithms on Haswell architecture using the AVX2 extension with FMA instructions shown as percentage against the use of standard AVX instructions in In8.

slightly increases the power consumption of the algorithm. The energy consumption values for this scenario show that the energy consumption is reduced by the use of the FMA instructions. This implies that the faster execution of FMA instructions saves slightly more time than energy.

C. *Effect of vectorization techniques on power consumption*

When comparing the auto-vectorization and intrinsic implementations against the sequential implementations there are multiple interesting differences. In Figure 4, it is clearly visible that the auto-vectorized implementation Av8 consumes less power than the sequential version. For the Sandy Bridge architecture, the intrinsic implementation In8 consumes even less power than the auto-vectorized implementation. For the Haswell architecture, the intrinsic algorithm In8 consumes nearly as much power as the sequential implementation. This difference may result because the algorithm in Listing 2 is not fully optimized and the compiler can be able to apply additional optimizations.

In comparison, the block-wise algorithms in Figure 5 shows that the intrinsic implementation In24 consumes the least power. In this case, for the auto-vectorized implementation Av24 on the Haswell architecture, there is still a power saving against the sequential version. On the Sandy Bridge architecture, the auto-vectorized implementation Av24 is consuming even more power than the sequential implementation Seq24. This effect might indicate, that the different optimizations that are applied by the compiler might also influence the power consumption with a negative effect.

In summary, it is possible to reduce the power consumption of an algorithm by up to 4% when using auto-vectorization on the Haswell architecture. Even more power can be saved if the implementation is done by using intrinsic functions for a manual vectorization. This saving can be up to 10% for both architectures.However, these findings strongly depend on the architecture.

D. *Influence of Loop Unrolling on power consumption*

The Figures 4 and 5 already provide the measurement data for analyzing the effect of loop unrolling on the power consumption of a program. In general, applying a higher stage of loop unrolling reduces the power consumption of a sequential algorithm.

For the auto-vectorized program versions, the differences between different loop unrolling stages are lower than those for the sequential algorithms. When executing on the Sandy Bridge architecture, the obtained results are nearly the same for all unrolling stages.

The power consumption of the intrinsic program versions with loop unrolling behaves qualitatively similar to the sequential and auto-vectorized ones. For the Haswell architecture, a higher loop unrolling stage also implies more power saving. This is mostly visible with higher frequencies. In total, using the block-wise 3x24 unrolling (In24) for these loops can save up to 8% of total power consumption compared to the basic program version. In contrast, the execution on the Sandy Bridge architecture consumes up to 6% more power with higher loop unrolling stages.

In general, it can be stated that the use of loop unrolling can reduce the power consumption of the sequential and vectorized algorithms.

E. *Influence of varying matrix sizes on power consumption*

A variable input size for the implementations is the matrix size. Although the algorithms are capable of taking any combination of size parameters, we have chosen quadratic matrices of the same extension for our experiments. This avoids of padding or inhomogeneous loop runs and, thus, makes obtained results clearer. Since all parameters l, m and n are identical, they are depicted as one value (i.e. $a \times a \cdot a \times a$ matrix size is simply shown as a).

The matrix size can influence the obtained results in different ways. First, a matrix size that is not fully dividable by the number of elements used in one iteration of the innermost loop can lead to the need for padding. To avoid this, we have chosen to use a stride of 48 for possible input matrix sizes. This ensures that in none of the loop unrolling stages, there is a need for padding, since 48 is dividable by 1 and 8, 2 and 16, and 3 and 24.

Choosing a matrix size that is small, reduces the runtime of the matrix multiplication but also increases fluctuations in measurement results. Choosing a matrix size that is large, reduces the size of fluctuations but increases the runtime and the probability of interference from the operating system. For that reasons we have empirically chosen the matrix size 3360, which is as small as possible but still delivers steady measurement results.

F. *Effect of optimization levels on power consumption*

For the measurements of the influence of compiler optimization levels on the power consumption, we have used

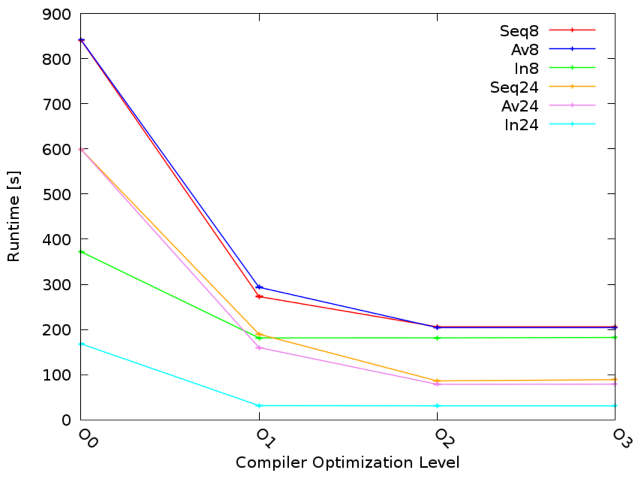

Figure 7: Runtimes of sequential, auto-vectorized and intrinsic program versions on Haswell architecture with frequency governor *powersave* for different compiler optimization levels.

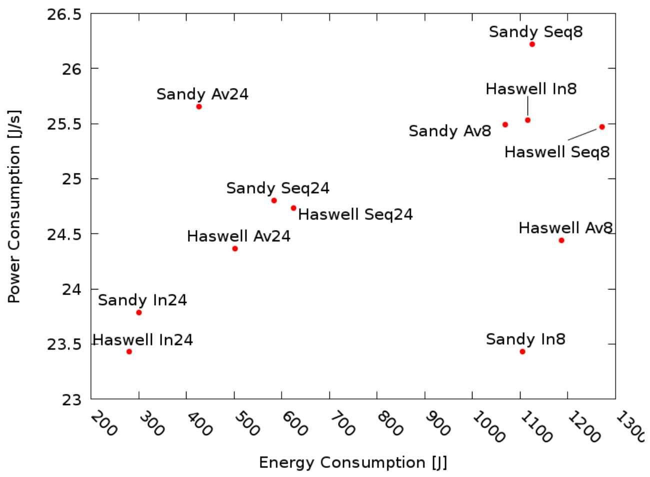

Figure 8: Power consumption of different program versions on different hardware architectures depicted against their respective energy consumption. the program versions Seq, Av and In in their unrolled stages 1x8 and 3x24 executed on a fixed frequency of 3.5 GHz on the Haswell architecture and 3.7 GHz on the Sandy Bridge architecture.

the dynamic frequency controller *powersave*. The *powersave* governor controls the frequency of the processor during the runtime of the program to consume the least power. Figure 7 shows that the runtimes of the program execution for different compiler optimization levels are longer for a lower optimization level and shorter for higher optimization levels. Additionally, the influence of the optimization level on the power consumption of the specific program version is very small. This leads to the conclusion, that the gained difference in power saving is vanishingly small and does not influence the obtained results. For the experiments shown before, we have used the compiler optimization level O3.

G. Summary

To depict optimal combinations of the program versions shown before, Figure 8 shows the power consumption of the different versions against their respective energy consumption. This enables the direct comparison of the power and energy consumption properties. Especially, Figure 8 enables to search for a program version that delivers the best trade-off for ones special needs. Since the runtime of a program also increases from left to right, a program delivering a good performance is located left. Additionally, a program consuming less power is located on the lower end of the figure. As an example for this the Haswell In24 execution consumes nearly the same amount of power as the Sandy Bridge In8, while delivering higher performance and consuming less energy. Another example might be the comparison of Haswell Av24 and Sandy Bridge Av24: The Haswell execution consumes less power, but the Sandy Bridge execution consumes less energy. It is clear that the Haswell In24 execution consumes the least power and energy.

IV. Related Work

In recent times there has been much research on energy efficiency of software. Especially, models for predicting the power consumption for the execution of different programming models have been introduced [5]–[7]. The focus of these studies mainly lies on the execution of parallel and concurrent programs. In our article, the parallelism is studied in the form of the SIMD vector instructions.

Ibrahim et al. have shown in [8] that vectorization can influence the power consumption of an embedded system based on a chip of Texas Instruments. However, they mainly focused on the usage of different compiler optimizations and their influence on the power consumption. On top of that, they rechecked their results against a vectorized code version. In our work, we are focusing on general purpose processors and the direct impact of vectorization which produces deviating results.

A brief analysis of the influence of vector operations on general purpose CPUs was done by Cebrián et al. in [9]. Additionally, they extended their research on further hardware platforms in [10]. They used the PARSEC-benchmarks *blackscholes* and *streamcluster* for their measurements. These benchmarks showed the capability of saving power when vectorizing them with intrinsics. For our research, we wanted to isolate the vectorization from concurrency to further investigate the influence of different optimizations and environment features. For this reason, we are using a handwritten algorithm, which we could modify easily.

Based on the performance of auto-vectorization in comparison to intrinsic vectorization, there are articles by Intel® [1], [2], which show that the auto-vectorization of the compiler is not capable of reaching the full performance which can be achieved by intrinsic programming. This performance gap can be narrowed by using special programming paradigms to make the code easier understandable for the compiler. We have evaluated if this performance gap also applies for the power consumption and could show that the performance gap is directly visible in power consumption, too.

V. Conclusion

In this article, we have investigated the influence of vectorization techniques on energy and power consumption of dense matrix algorithms. We inspected the interrelation between vectorization, compiler optimization levels, loop unrolling, and other environmental influences, such as frequency, input data size, and processor architecture. The energy consumption of a program can be easily reduced using performance aimed optimizations and executing the program on a mid-level processor frequency. This results out of the shorter execution time which is directly reflected in energy saving.

We also have been able to show, that vectorization provides the ability to reduce the power consumption of a program if it is implemented by using loop unrolling and frequency control. In general, the power consumption of both vectorized and sequential programs can be reduced by up to 70%, when running on the lowest possible processor frequency. In many cases, it is possible to reduce the power consumption of algorithms with the use of vectorization of up to 10% compared to a similar sequential version for the same frequency. The manual vectorization of an algorithm using compiler intrinsics saves more power than auto-vectorization done by the compiler. The usage of loop unrolling can alter the power consumption of an algorithm both to the better or to the worse. The use of different compiler optimization levels in our research did not lead to a notable effect on power consumption. However, the utilization of different hardware architectures results in up to 10% lower power consumption values.

When applying the results of this article, one has to take special care that the considered effects are not fully independent and the list of influential factors is non-exhaustive.

Acknowledgment

This work was performed within the Federal Cluster of Excellence EXC 1075 "MERGE Technologies for Multifunctional Lightweight Structures" and supported by the German Research Foundation (DFG). Financial support is gratefully acknowledged.

References

[1] C. Kim, N. Satish, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology," 10 2013. [Online]. Available: http://www.intel.com.br/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-closing-ninja-gap-paper.pdf

[2] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture*.

[3] Intel® Corporation, *User and Reference Guide for the Intel C++ Compiler 15.0*, 3 2015. [Online]. Available: https://software.intel.com/en-us/compiler_15.0_ug_c

[4] ——, *Intel® 64 and IA-32 Architectures Software Developers Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, 9 2014. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[5] T. Rauber, G. Rünger, and M. Schwind, "Energy Measurement and Prediction for Multi-Threaded Programs," in *22nd High Performance Computing Symposium 2014 (HPC 2014), Part of the SCS Spring Simulation Conference*, 2014.

[6] T. Rauber, G. Rünger, M. Schwind, H. Xu, and S. Melzner, "Energy measurement, modeling, and prediction for processors with frequency scaling," 2014.

[7] T. Rauber and G. Rünger, "Modeling and Analyzing the Energy Consumption of Fork-Join-based Task Parallel Programs," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 1.

[8] M. Ibrahim, M. Rupp, and A. Fahmy, "Code transformations and SIMD impact on embedded software energy/power consumption," in *International Conference on Computer Engineering Systems, 2009. ICCES 2009.*, 2009.

[9] J. Cebrián, L. Natvig, and J. Meyer, "Improving Energy Efficiency through Parallelization and Vectorization on Intel Core i5 and i7 Processors," in *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, 11 2012.

[10] ——, "Performance and energy impact of parallelization and vectorization techniques in modern microprocessors," *Computing*, vol. 96, no. 12, 2014.