Proceedings of the Federated Conference on Computer Science and Information Systems pp. 319–327 DOI: 10.15439/2018F28 ISSN 2300-5963 ACSIS, Vol. 15

On the energy consumption of Load/Store AVX instructions

Thomas Jakobs, Gudula Rünger
Chemnitz University of Technology,
Department of Computer Science
09111 Chemnitz, Germany
E-mail: [thomas.jakobs, ruenger]@cs.tu-chemnitz.de

***Abstract*—The energy efficiency of program executions is an active research field in recent years and the influence of different programming styles on the energy consumption is part of the research effort. In this article, we concentrate on SIMD programming and study the effect of vectorization on performance as well as on power and energy consumption. Especially, SIMD programs using AVX instructions are considered and the focus is on the AVX load and store instruction set. Several semantically similar but different load and store instructions are selected and are used to build different program versions of for the same algorithm. As example application, the Gaussian elimination has been chosen due to its interesting feature of using arrays of varying length in each factorization step. Five different SIMD program versions of the Gaussian elimination have been implemented, each of which uses different load and store instructions. Performance, power, and energy measurements for all program versions are provided for the Intel Sandy Bridge, Haswell and Skylake architectures and the results are discussed and analyzed.**

***Index Terms*—Energy consumption, power consumption, AVX instruction set, Gaussian elimination.**

I. Introduction

In addition to performance optimization of codes for numerical algorithms, there is a growing need to optimize the power and energy consumption of such programs as well. Reasons are well-known and include aspects, such as budgeting, battery life, cooling capacity, or physical boundaries. A possibility to reduce the power and energy consumption of the execution of a program is the choice of energy-saving architectures and or architecture components, which have been developed by the hardware manufacturers for this purpose. An example for such an architecture components are SIMD or vectorization units supplied by recent processors. A CPU based SIMD execution unit calculates multiple elements in special registers simultaneously with one instruction in contrast to processing them sequentially. For the same computation, a vectorized program activates less transistors than a sequential program and, hence, vectorization provides a potential for performance and, power and energy optimization.

The exploitation of SIMD units for executing numerical algorithms requires to provide a suitable program which contains so-called vector operations that cause the architecture to exploit the vector units. One possibility to do this is to use the Intel AVX instruction set, which can be used on recent Intel architectures, such as the Intel processors Sandy Bridge, Haswell or Skylake. The design of such a vectorized program using AVX instruction includes several decisions when transforming a sequential program or algorithm into a vectorized program version. The decisions include the selection of program parts to be coded as vector operations but also the specific choice of AVX instructions to be used for the coding. Both, the strategy to include SIMD parallelization into a program as well as the choice of instructions, can have an effect on the performance as well as the power and energy consumption. This article concentrates on the AVX load and store instruction set and investigates the effect on the performance when different but semantically similar instructions are chosen.

The investigations of this article concentrate on different load and store instructions provided by the AVX instruction set, such as aligned, unaligned, streaming or masked load and store instructions. Using these alternatives for load and store, different SIMD program versions for the Gaussian elimination are built. The experimental investigation of these program versions exhibits interesting differences in the performance results, such as the different performance of aligned and unaligned load and store instructions. In detail, this article makes the following contributions:

- Several program versions for the Gaussian elimination are implemented with different AVX instructions.
- The performance, power and energy of the the five program versions with different AVX instructions have been investigated on three processor architectures.
- A detailed discussion and comparison of differences in performance, energy and power consumption of the implemented program versions is given.

The rest of this article is structured as follows: In Sec. II, we introduce AVX instructions, the Gaussian elimination, and the vectorized program versions of the Gaussian elimination. Section III introduces the execution environment and hardware architecture. Section IV presents the performance properties of the program versions. In Sec. V, we discuss the influence of different AVX instruction on the energy efficiency of a program. Section VI shows related research and Sec. VII concludes.

II. SIMD IMPLEMENTATION OF THE GAUSSIAN ELIMINATION

The vectorized implementation of the Gaussian elimination leaves room for different choices of load/store AVX instructions. This section introduces the AVX instructions, their expected influence, the Gaussian elimination, and the vectorized program versions.

A. Programming with AVX

Many of today's CPUs have vector or SIMD instruction sets, which support parallel executions by applying the same operation simultaneously to two, four, or more pieces of data. Two popular SIMD instruction sets are the *Streaming SIMD Extensions* (SSE) and the *Advanced Vector Extensions* (AVX), both implemented into various AMD® and Intel® processors. SSE and AVX provide instructions to load, modify, and store 128-bit (SSE) or 256-bit (AVX) vectors containing multiple elements, where the number of elements is specified by their particular size. In principle, program executions can reach a speedup equal to the number of data elements per vector, in practice several factors limit the optimal speedup. The performance properties and limitations for vectorization have been demonstrated for multiple examples, e.g. in [2]. Only few articles cover vectorization in the context of energy efficiency. Examples from Cebrián et. al. [3] or Lorenz et. al. [4] demonstrate a potential to increase the energy efficiency by the application of vectorization.

The SSE and AVX instructions can be used either as assembler instructions or embedded into C-style intrinsic functions. It is recommended to use intrinsic functions instead of assembler instructions, since it enables the compiler to apply further optimizations, such as dead code analysis. The intrinsic functions have the following format:

```
<type>
_mm<size>_<operation>_<type_suffix>(
<type> param1,..)
```

- `<type>` can either be a standard C-type (e.g. `float`) or a special vector type (e.g. `__m256` for 8 single-precision floating-point values) depending on the actual function purpose and definition.
- `<size>` denotes the number of bits used for the instruction, e.g. `256` for AVX.
- `<operation>` expresses the implemented operation, e.g. `add` for an addition.
- `<type_suffix>` denotes the type of data to operate on, e.g. `ps` for packed single precision.
- The number and type of parameters varies dependent on the instruction.

Table 1 lists the specific intrinsic functions used in this article with their semantic, and the values for latency and throughput on different processor architectures given in [1].

B. Alternative load/store instructions

The different semantic of the intrinsic functions can be investigated with regard to the energy efficiency. For this purpose the intrinsic functions in Tab. 1 are evaluated in the context of expected differences in measurement results. Such differences can arise from their difference in latency and throughput, and due to their requirement on memory alignment, i.e. the memory address being divisible by 32 byte. The following instructions are amenable to demonstrate such differences:

- **Unaligned** load/store: Load and store operations accessing unaligned memory are applicable for any memory position to load/store any consecutively stored elements. The cache line size is a multiple of the AVX register size of 256 bit, and thus the access of unaligned memory positions may contain a cache line border, resulting in a possible performance loss [5].
- **Aligned** load/store: Vectors aligned at 256 bit reside in the same cache line. Thus, the access is expected to lead to a higher performance of the program, compared to unaligned instructions. However, an aligned memory access has to be ensured by the programmer using methods such as loop peeling, special allocators, and/or alternative strategies.
- **Streaming** store: Streaming stores are special, aligned store instructions that bypass the caches when storing data. In detail, they are implemented using a "non-temporal hint" to indicate that no intermediate copy should be created in cache. Thus, streaming stores provide the possibility to issue a Write Through operation at the programming level.
- **Masked** load/store: Vector instructions usually use all elements for their calculations leading to a strict SIMD implementation and execution. A possibility to use only parts of a vector register is provided by masked instructions. Masked load/store instructions have an additional mask parameter which specifies the elements to be loaded/stored. The elements to be omitted, i.e. not to be loaded/stored, are specified by 0-bits in the mask parameter. Values omitted by loads are assigned zeros in the vector, whereas values omitted by stores are skipped while writing to memory. Values to be loaded/stored are identified by 1-bits in the mask parameter and treated accordingly.

```
for k = 0 < N-1; k+=1
  //exchange rows with pivot element
  for i = k+1 < N; i+=1
    L[i*N+k] = A[i*N+k] / A[k*N+k]
    for j = k+1 < N; j+=1
      A[i*N+j] = A[i*N+j] - A[k*N+j] * L[i*N+k]
    b[i] = b[i] - b[k] * L[i*N+k]
//Backward substitution
```

Listing 1: Algorithm of a Gaussian elimination adapted from [6] of which the vectorized program versions are derived.

C. The Gaussian elimination

The Gaussian elimination is an important kernel in scientific applications for solving linear equations. For the Gaussian elimination, a set of N linear equations with N

Intrinsic function	Instruction semantic	HSW		SKL	
		Lat	Tp	Lat	Tp
`_mm256_load_ps(*mem)`	Loads 8 float values from an aligned memory position `mem` into a vector variable.	1	0.5	1	0.25
`_mm256_loadu_ps(*mem)`	Loads 8 float values from an unaligned memory position `mem` into a vector variable.	1	0.5	1	0.25
`_mm256_maskload_ps(*mem,mask)`	Loads a specified selection (`mask`) of values from a memory position `mem` into a vector variable. Omitted values are 0.	8	2	11	1
`_mm256_broadcast_ss(*mem)`	Loads one float value (`mem`) into all elements of a vector variable.	-	-	-	-
`_mm256_loadu_si256(*mem)`	Loads 256 bits of integer values from an unaligned memory position `mem` into a vector variable.	1	0.25	1	0.25
`_mm256_store_ps(*mem,a)`	Stores the elements of a vector variable (`a`) into an aligned memory position `mem`.	1	0.5	1	0.25
`_mm256_storeu_ps(*mem,a)`	Stores the elements of a vector variable (`a`) into an unaligned memory position `mem`.	1	0.5	1	0.25
`_mm256_maskstore_ps(*mem,mask,a)`	Stores a specified selection (`mask`) of values from a vector variable `a` into a memory position `mem`. Omitted values are skipped.	-	2	-	1
`_mm256_stream_ps(*mem,a)`	Stores the elements of a vector variable (`a`) into an aligned memory position `mem` using a non-temporal hint.	-	1	-	1
`_mm256_mul_ps(a,b)`	Multiplies the elements of two vector variables (`a` and `b`) with each other.	5	0.5	4	0.5
`_mm256_sub_ps(a,b)`	Subtracts the elements of one vector variable (`b`) from another vector variable (`a`).	3	1	4	0.5

Table 1: AVX intrinsic functions used in this article with their respective values for Latency (Lat.) and Throughput (Tp.) for the Haswell (HSW) and Skylake (SKL) architectures from [1].

unknowns x_k and their respective coefficients a_{ik} and right hand side b_i, where $1 \leq i,k \leq N$ is given. The equation and solution for $Ax = b$, with $A \in \mathbb{R}^{N\times N}$ and $x, b \in \mathbb{R}^N$, has to be solved.

The Gaussian elimination can be divided into two phases: A forward elimination and a backward substitution. In the forward elimination the matrix A is transformed into an upper triangular form, such that $Ux = b'$ holds, where U is the matrix in upper triangular from.

The forward elimination is depicted in Listing 1 as pseudocode. The *k-loop* in Line 1 executes the steps of the forward elimination and iterates along the diagonal elements a_{kk} of matrix A. Each step starts with a pivot search in which the maximum value below (a_{ik}) the diagonal element a_{kk} is searched, and a row exchange of the *current* row $a_{k\_}$ with the row of the pivot $a_{piv\_}$ is done. In each step the *i-loop* in Line 3 calculates for each row $a_{i\_}$, with $i > k$, the elimination factor $l_{ik} = \frac{a_{ik}}{a_{kk}}$ (Line 4), which is stored in a separate matrix $L \in \mathbb{R}^{N\times N}$. Using the elimination factor l_{ik} all elements of row $a_{i\_}$ are calculated by $a_{ij} = a_{ij} - a_{kj} \cdot l_{ik}$, where $k+1 \leq j \leq N$ denotes the column that is iterated by the loop in Line 5. Afterwards, for each row the element b_i of the result vector b is calculated by $b_i = b_i - b_k \cdot l_{ik}$. After $N-1$ steps the former matrix A is transformed to upper triangular form U. The resulting matrix U is stored in the same memory location as the former matrix A, in which the lower triangular elements are *assumed* to be zero.

The second phase is the backward substitution in which the values for vector x are calculated. The elements of vector x are calculated in the order of $x_N, x_{N-1}, \ldots x_1$ according to $x_k = \frac{1}{a_{kk}}\left(b_k - \sum_{j=k+1}^{N} a_{kj} \cdot x_j\right)$.

The algorithm of Listing 1 is not optimized to preserve the ratio of memory- and computation-instructions.

```
1  for k = 0 < N; k+=1
2    //exchange rows with pivot element
3    for i = k + 1 < N; i+=1
4      L[k*N+i] = A[i*N+k] / A[k*N+k];
5      __m256 lv = _mm256_broadcast_ss(&L[k*N+i]);
6      for j = k + 1 < N - 8; j+=8
7        __m256 ak = _mm256_loadu_ps(&A[k*N+j]);
8        __m256 ai = _mm256_loadu_ps(&A[i*N+j]);
9        ak = _mm256_mul_ps(ak,lv);
10       ai = _mm256_sub_ps(ai,ak);
11       _mm256_storeu_ps(&A[i*N+j],ai);
12     if(j < N)
13       int loadmask = {0,0,0,0,0,0,0,0,
           -1,-1,-1,-1,-1,-1,-1,-1};
14       __m256i mask = _mm256_loadu_si256(
           (__m256i*)&loadmask[N-j]);
15       j = N - 8;
16       __m256 ak = _mm256_loadu_ps(&A[k*N+j]);
17       __m256 ai = _mm256_loadu_ps(&A[i*N+j]);
18       ak = _mm256_mul_ps(ak,lv);
19       ai = _mm256_sub_ps(ai,ak);
20       _mm256_maskstore_ps(&A[i*N+j],mask,ai);
21   //calculate b similarly
22 //Backward substitution
```

Listing 2: Vectorized implementation of the Gaussian elimination from Listing 1, which is the starting point for the subsequent program versions.

D. SIMD implementation versions of the Gaussian elimination

A vectorized implementation of the Gaussian elimination from Listing 1 is presented in Listing 2. The implementation uses the intrinsic functions introduced in Tab. 1 to calculate multiple elements of the row (of the *j-loop*) simultaneously. Lines 1 to 4 of Listing 1 remain unchanged. For the vectorized calculation the resulting value of l_{ik} is copied into all elements of a vector variable `lv` in Line 4. The *j-loop* in Line 6 is unrolled eight times to calculate

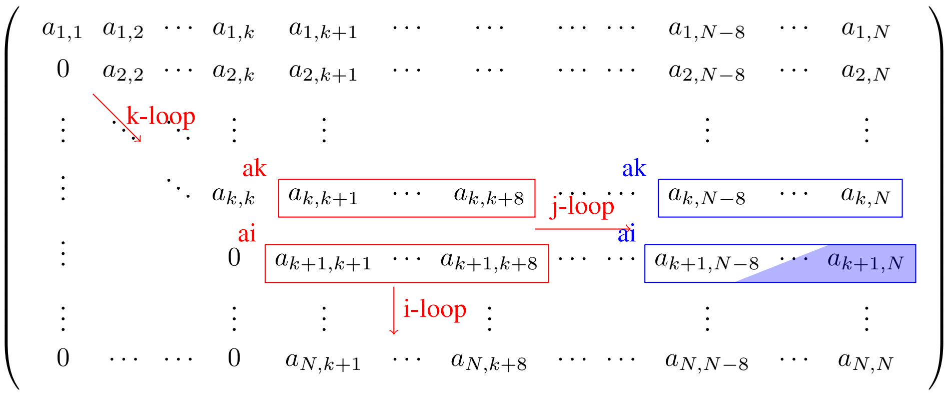

Figure 1: Matrix A in step k, with $i = k + 1$ and $j = k + 1$. The vector variables and the iteration directions according to Listing 2 are depicted in red. The remainder is calculated using the vectors highlighted as blue rectangles for which only the new elements (blue filled) are stored back to the array.

eight single-precision floating-point values simultaneously as demonstrated in Fig. 1. For the use with vector instructions the elements are loaded into vector variables (`ak` and `ai` in Lines 7 and 8, red rectangles in Fig. 1). The vector variables are used to calculate the new values for $a_{i,j}, a_{i,j+1}, \ldots, a_{i,j+7}$ (see red `ai` in Fig. 1) that are written back into the array in Line 11.

Since the number of iterations for the *j-loop* $(N-(k+1))$ is not guaranteed to be divisible by eight a special case has to be implemented. The remaining elements are handled in the block after Line 12 using the masked instructions. Usually, masked instructions are used to load the last elements of the row and values which would not be part of the row are omitted. We choose a different strategy to avoid two costly `maskload` instructions. For this strategy the last eight elements of each row are loaded, regardless of how many are actually needed (see blue rectangle in Fig. 1). After calculation, when storing the results, a mask is applied to write only those elements that are part of the remainder, discarding the twice used elements (blue fill of `ai` in Fig. 1). The mask is created by using an array (`loadmask` in Line 13) of 256 0-bits followed by 256 1-bits and loading the mask in Line 14 which contains the correct amount of 1-bits. The calculation of the $N-(k+1)$ elements of vector b is done similarly.

Different program versions of the Gaussian elimination are built with different load/store instructions. In the following we describe the five program versions:

- The **storeu** program version is the *starting point* implementation using unaligned loads and stores. The *storeu* program version is presented in Listing 2.
- The **store** program version uses aligned loads (`_mm256_load_ps`) and stores (`_mm256_store_ps`). Hence, the *j-loop* in Line 6 does not start at $j = k + 1$ but at the beginning of the aligned block of memory in which $k + 1$ resides. Thus, additional elements are calculated, but a peeling loop is avoided. The aligned load instructions are implemented in Line 8, and the aligned store instructions in Line 11 of Listing 2.
- The **stream** program version uses aligned loads identically to the *store* program version. However, the `store` instruction in Line 11 is replaced with a streaming store (`_mm256_stream_ps`) instruction which bypasses the cache while writing.
- The **maskload** program version replaces all load instructions with masked load (`_mm256_maskload_ps`) instructions. Explicitly, the `maskload` instruction is implemented in Lines 7, 8, 16 and 17 of Listing 2. The masks of these may be either all 1-bits or the correct mask for the remainder. The *maskload* program version illustrates the difference between masked loads and normal loads.
- The **SeqRem** program version replaces the vectorized remainder (Lines 12 to 20 in Listing 2) with a sequential loop that processes each remaining element sequentially as in Listing 1. A sequential remainder loop may be more efficient due to the expected higher cost of masked instructions.

The program versions cover a set of selectable instructions to implement a vectorized Gaussian elimination. The program versions differ only in the modifications shown.

III. Experimental Evaluation

The measurements for this article are conducted in the environment described in this section. Each measurement is conducted ten times and the presented values are averaged. For the measurements we use a matrix A that has $10\,000 \times 10\,000$ elements.

A. Execution Environment

For the measurements of this article we use three Intel® Core i7 processors with a similar specification but different architecture families. The three processors are: A Core i7-2600 of the Sandy Bridge architecture, a Core i7-4770K of the Haswell architecture, and a Core i7-6700 of the

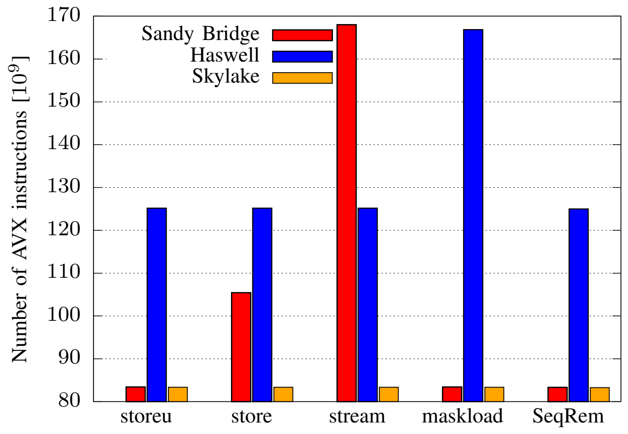

Figure 2: Number of AVX instructions actually executed in the CPU by the execution of the five program versions on the three processors.

Skylake architecture. All three processors allow processor frequency scaling with the `cpu-freq` tool. For the Haswell and Skylake processor scaling is enabled between 0.8GHz and 3.5GHz and for the Sandy Bridge processor the scaling is possible between 1.6GHz and 3.7GHz.

We measure the energy consumption of program executions using a simple interface to read the on chip energy values provided by the Intel® RAPL interface. Additionally, we measure other performance counters using the PAPI library version 5.5. We compiled the program versions using the Intel® C++ Compiler (`icc` version 17.0.0 [gcc version 4.9.0]) with the additional compiler flags `-O3` and `-restrict`. Additionally, we applied the compiler flags `-mavx` for Sandy Bridge architecture and `-march=core-avx2` for the other architectures. The usage of the `-march-avx2` flag implies the usage of FMA instructions rather than `sub` and `mul` by the compiler.

B. Issued AVX instructions

A variation of the number of issued AVX instructions occurs when executing the five program versions on the three processors. The number of issued AVX instructions can be measured during program execution and reflects the number of 256-bit instructions executed.

The number of issued AVX instructions is depicted in Fig. 2 for the different program versions from Sec. II and the three architectures. It is notable that most of the program executions have a nearly identical number of issued instructions. The expected behavior would be all issued instruction counts being nearly identical, since the number of instructions should be predefined by the number of assembler instructions that are generated by the intrinsic functions.

There are multiple exceptions to the expected behavior. For all program versions, the number of instructions issued on the Haswell architecture is about 50% higher, than for the other two architectures. Additionally, some program versions generate an untypical high number of executed

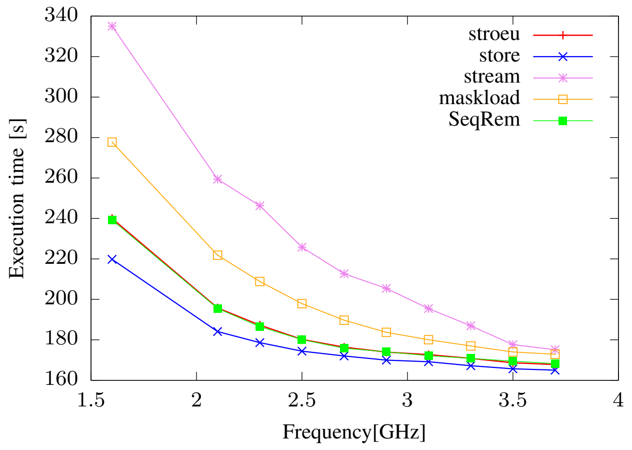

Figure 3: Execution time of the Gaussian elimination versions from Sec. II on Sandy Bridge architecture depending on CPU frequency.

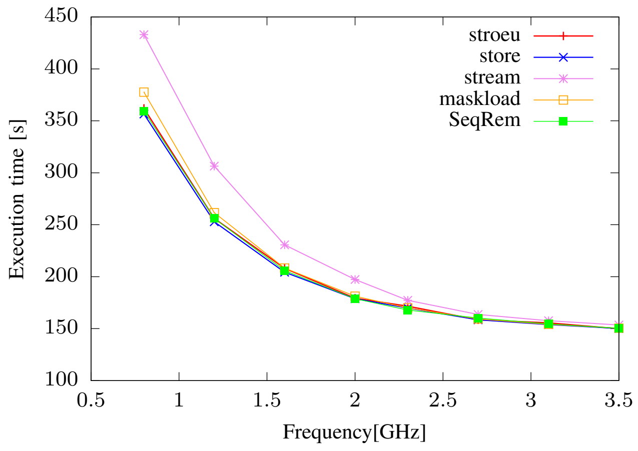

Figure 4: Execution time of the Gaussian elimination versions from Sec. II on Haswell architecture depending on CPU frequency.

instructions. These exceptions arise at the execution of the program version with a higher amount of `maskload` instructions for the Haswell architecture, the use of the `stream` instruction on the Sandy Bridge architecture and slightly more with the use of the aligned load and store operations for the Sandy Bridge architecture.

IV. Performance Evaluation of the Program Versions

In this section, we present the measurements and discuss the differences in execution time for the different program versions. The five program versions of Sec. II are executed on the three processors described in Sec. III.

A. Evaluating execution time

The Fig. 3, 4 and 5 display the execution times of the different program versions in dependence to the processor frequency. As expected, a higher processor frequency leads in all diagrams to a shorter execution time and the qualitative

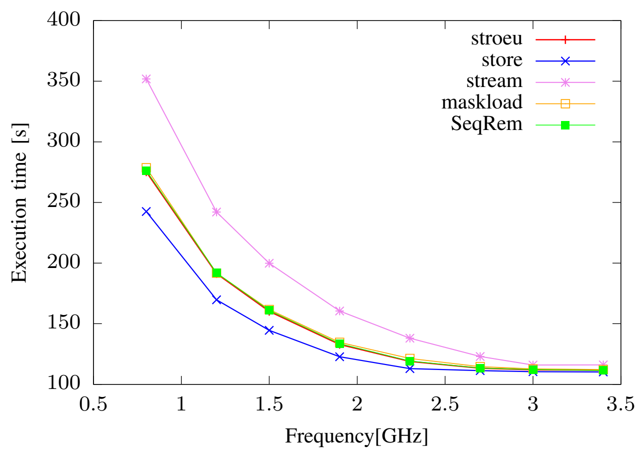

Figure 5: Execution time of the Gaussian elimination versions from Sec. II on Skylake architecture depending on CPU frequency.

behavior is similar. However, the effects of different AVX instructions on performance are thus revealed by the relation of the execution time of the program versions to each other. The program version using the `stream` instruction has the longest execution time for all architectures. This might be caused by the Write Through strategy that is applied with the `stream` instruction which bypasses the cache when writing data. Thus, the processor has to wait for completion of the write operation (*write-wait*) before another `stream` instruction can be executed.

The *write-waits* are a significant problem occurring for the Gaussian elimination, which has a high number of loads and stores with only little computation per element in between. In other algorithms, that implement a lower number of streaming stores the *write-waits* may be hidden by other computations, loads or Write Back stores.

The Sandy Bridge architecture was the first architecture to support AVX instructions and many improvements to hardware for AVX have been made since. This is also reflected in the example of the program version executing a higher number of `maskload` instructions. For the Sandy Bridge architecture the *maskload* program version takes a higher execution time than the remaining three program versions. For the Haswell architecture the execution time is only slightly higher and for the Skylake architecture there is no difference in execution time between a `maskload` (with a full true-mask) and a `loadu` instruction.

A comparison between the *storeu* program version and the *SeqRem* program version demonstrates the influence of a vectorized remainder against a sequential remainder loop. The measurement results of the *storeu* and the *SeqRem* program versions are nearly identical ($< 0.5\%$). In combination with the results of the *maskload* program version, this leads to the conclusion, that the use of masked instructions has a worse performance than regular load/store operations, but are at least as performant as a sequential program execution.

For the Sandy Bridge architecture the aligned `load` and `store` instructions exhibit a better performance than the unaligned ones. The difference between aligned and unaligned instructions is eliminated for the Haswell architecture. For the Skylake architecture the performance enhancement of aligned instructions is again visible. Presumably, the architectural changes between Sandy Bridge and Haswell architecture improved the unaligned instructions, whereas the architectural improvement from Haswell to Skylake architecture improved the aligned instructions.

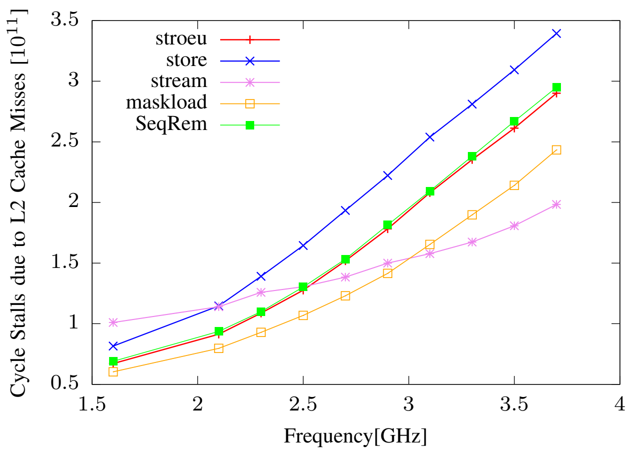

Figure 6: Cycle Stalls due to level 2 cache misses of the Gaussian elimination versions from Sec. II on Sandy Bridge architecture depending on CPU frequency.

B. Influence of Cache-Waits on execution time

The memory properties of a program can support the classification of its performance properties. One way to get information about the memory properties is to take a look at the cache usage of the program versions.

Figure 6 displays the number of CPU cycles stalled due to waits for pending operations on level 2 cache for the Sandy Bridge architecture. For most of the program versions from Sec. II the relation of the curves is directly inverse to their execution time of Fig. 3. This behavior is expected due to the limitation of memory bandwidth that gets visible more clearly if the program executes the implemented calculations faster.

The *stream* program version produces a lower rise in cycle stalls in dependence to the processor frequency than the other four program versions. The lower rise in the diagram can be explained with the operation inside the `stream` instruction: The Write Trough operation. The Write Trough operation does not write data into the cache but directly to main memory. Thus, waiting for a write operation is not counted as a wait for any cache and thus does not get counted for level 2 cache stalls. When regarding other resource stalls, i.e. general stalls, the results for the *stream* program version are much higher than for the other program versions. This observation reinforces the previous statement of the worse execution time of the `stream` instruction resulting from *write-waits*. The difference is presented for the Sandy Bridge architecture in Fig. 6 but is also observ-

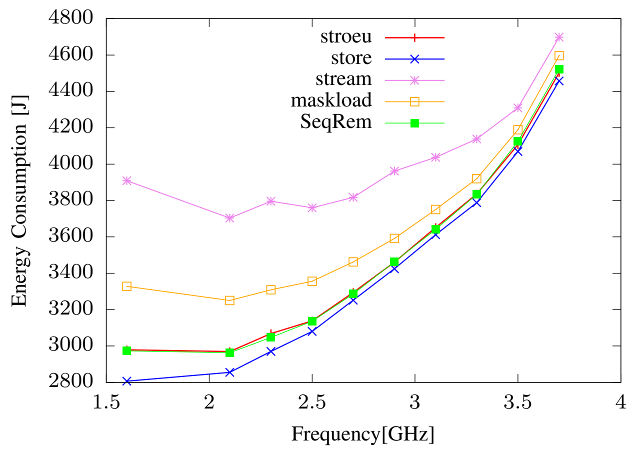

Figure 7: Energy consumption of the Gaussian elimination versions from Sec. II on Sandy Bridge architecture depending on CPU frequency.

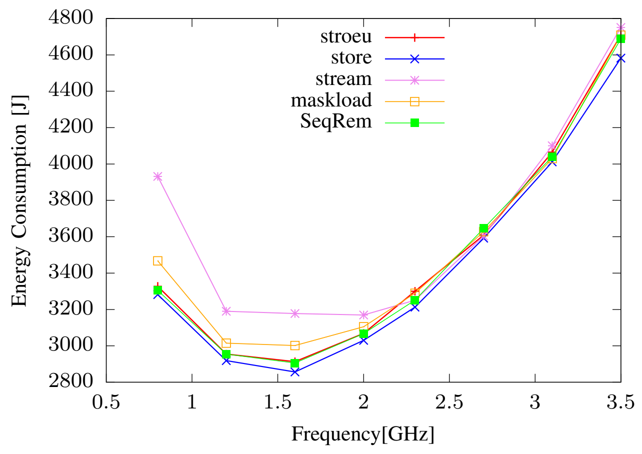

Figure 8: Energy consumption of the Gaussian elimination versions from Sec. II on Haswell architecture depending on CPU frequency.

able for the Haswell and Skylake architectures. Since the investigations for this article do not exploit parallelism, the level 3 cache displays a similar behavior.

V. Examining the Energy Efficiency

The main question of this article is the behavior of the energy efficiency of different AVX instructions. This section discusses the energy and power consumption of the five program versions from Sec. II. The results are presented as *Energy to Solution* and *Power to Solution* from which other metrics, such as Energy-Delay-Product, can be derived.

A. Evaluation of the energy consumption

One metric to discuss the energy efficiency of program execution is the energy consumed by the processor during the execution of the program version. The Fig. 7, 8 and 9 present the energy consumption of the program versions from Sec. II depending on the processor frequency. Similar

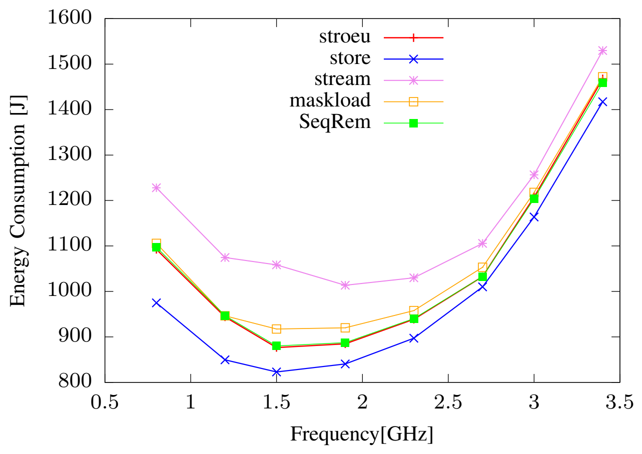

Figure 9: Energy consumption of the Gaussian elimination versions from Sec. II on Skylake architecture depending on CPU frequency.

to the results of the performance discussion, the *stream* program version has the highest (worst) energy consumption for all architectures. The *maskload* program version consumes more energy on the Sandy Bridge and Haswell architecture. Executing the remainder with a sequential remainder loop does not change the energy consumption against a vectorized remainder. In contrast to the performance discussion, the program version using aligned stores consumes less energy for all three architectures.

The lowest energy consumption is achieved with a frequency between 1.5GHz and 2.0GHz for all program versions and architectures. The dependency of the energy consumption on the processor frequency produces a U-Shape with deviations. Specifically, the U-Shape of the *stream* program version creates a U-Shape with a broader base, i.e. a flatter U-Shape. A similar behavior is shown by the *maskload* program version on the Haswell and Skylake architectures.

The different program versions produce their highest difference in energy consumption for the lower frequencies. For the higher frequencies the execution of the program versions is mostly affected by the memory transfer time, as already discussed in Sec. IV. The dependence on memory bandwidth limits the capabilities of vectorization and thus makes idle or waiting time a significant fraction of the program execution.

B. Differences in power consumption

In many cases the implementation with different vector instructions changes energy consumption in the same way as it changes the execution time of the program execution. Some exceptions can be found by regarding the power consumption of the program versions, where $power = \frac{energy}{time}$.

The power consumption is calculated from the averaged measurement results for execution time and energy consumption of each execution. The power consumption of a program execution strongly depends on the processor

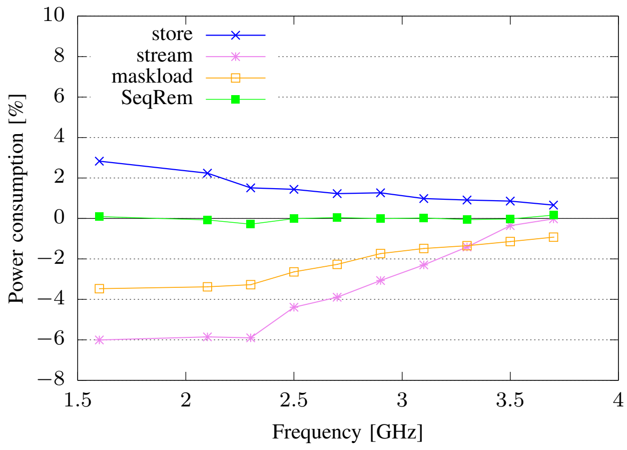

Figure 10: Difference of power consumption of the Gaussian elimination versions from Sec. II to the *storeu* program version in percent on Sandy Bridge architecture depending on CPU frequency.

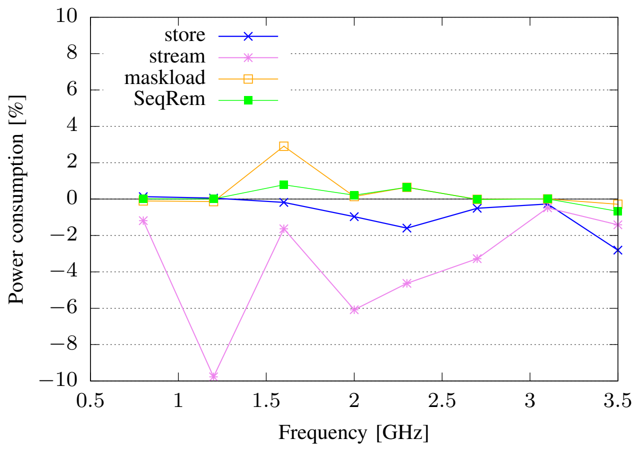

Figure 11: Difference of power consumption of the Gaussian elimination versions from Sec. II to the *storeu* program version in percent on Haswell architecture depending on CPU frequency.

frequency [7]. Thus, to discuss the differences between the five program versions the Fig. 10, 11 and 12 display the power consumption as difference from the *storeu* program version in percent. In general, a processor consumes more power when more transistors are active, i.e. the processor gets hotter, which often comes with a shorter execution time and less energy consumption.

Overall, the five program versions produce less differences for higher frequencies, which demonstrates the dependency on memory bandwidth rather than computing capabilities. Additionally, the sequential remainder loop program version has nearly no difference ($< 0.5\%$) to the *storeu* program version with a vectorized remainder.

The *stream* and *maskload* program versions produce a local maximum or peak behavior for the processor frequencies around 1.5GHz for the Haswell and Skylake architectures in Fig. 11 and 12. The peak results from the flatter U-

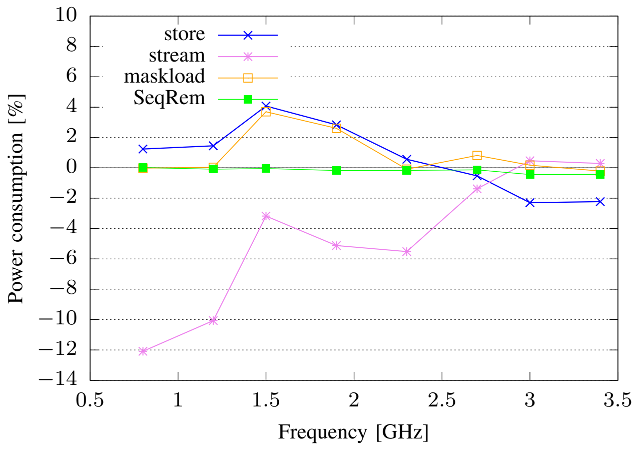

Figure 12: Difference of power consumption of the Gaussian elimination versions from Sec. II to the *storeu* program version in percent on Skylake architecture depending on CPU frequency.

Shape that these two program versions display in energy consumption, where the *storeu* program version (0%-Line) does not follow this behavior.

In Fig. 10 the *maskload* program version generates a lower power consumption for the Sandy Bridge architecture than the *storeu* program version. For the other two architectures the relation is directly inverse or no difference is shown at all. The reason for this is the higher energy consumption of the *maskload* program version on the Haswell and Skylake architecture for which the execution time is identical to the *storeu* program version. For the execution of the *maskload* program version on the Sandy Bridge architecture the execution time is higher (+16% for 1.6GHz) as well as the energy consumption is higher (+12% for 1.6GHz). The difference between these two increased values leads to a lower power consumption.

The program version implemented with aligned stores produces a lower energy consumption than the *storeu* program version for all architectures. However, the execution time is nearly identical on the Haswell architecture and lower on the other two architectures. This leads to a reduced power consumption of the *store* program version on the Haswell architecture. Additionally, for the Skylake architecture the *store* program version produces a constant difference in energy consumption and execution time to the *storeu* program version for higher frequencies. However, the point at which the constancy is occurring is at 2.0GHz for the energy consumption and at 2.7GHz for the execution time. This leads to an inverse relation of the *store* program version in Fig. 12 above 2.7GHz, for which the aligned stores are more power and energy efficient.

VI. Related Work

The energy and power efficiency is subject to different research fields. Especially, in the field of multi-threaded, parallel and distributed computing [3], [8], [9]. In our work we isolate the effects of vectorization on energy efficiency and leave other techniques of parallel execution applicable on top.

Lien et al. [10] investigate the energy efficiency of multi-threaded vectorized programs. They use three different algorithms for their work: FFTW, Matrix-Multiplication, and blackscholes. They show that vectorization increases energy efficiency, even more with additional multi-threading. For our investigations we isolate the use of vectorization to reason about the impact of different instructions used.

Caminal et al. present an energy efficiency study of the ParVec Suite based on different vectorization strategies [11]. They demonstrate the need for easy user guided vectorization to reduce the energy consumption of program executions. Their main focus is on the results of different user guided vectorization techniques such as OmpSs and Mercurium. The focus of our work is the investigation of differences of vector instructions to create additional knowledge for the use in such user guided vectorization systems.

In [2] Kim et al. describe modifications of source code to increase the performance properties of vectorized programs. They specifically emphasize that the use of continuously stored data elements is one of the key factors for efficient vectorized programs. For this they extensively discuss the use of Struct-of-Arrays instead of Array-of-Structs store order. We considered their findings for creating our program versions and focus on the influence of different instructions.

Hofmann et al. examine the influence on performance of vector instructions with the RabbitCT benchmark in [12]. They demonstrate that the choice of instructions has an influence on the performance of the program execution. Their work examines the additional instructions introduced by the AVX2 instruction set and Intel Xeon Phi instructions. We extend this research by examining the performance and energy efficiency for regular load/store instructions.

VII. Conclusion

In this article, the influence of different load and store AVX instructions with different program versions of a Gaussian elimination are investigated. The investigations demonstrate that the choice of instructions influences the execution time, energy and power consumption. The number of issued AVX instructions may be different, depending on the processor architecture and set of instructions implemented. However, no influence of a different number of issued AVX instructions on the performance or energy properties of the program execution can be derived.

The processor development influences the properties of different instructions, such as for the masked instructions which are improved in the newer processors. The use of streaming store instructions produces the worst behavior due to the memory intensiveness of the Gaussian elimination. Additionally, remainder loops can be vectorized without performance or energy efficiency being negatively influenced. Aligned load and store instructions produce the best results in performance and energy consumption even if additional elements are computed.

A next step in our research will be to identify the root cause of the different number of instructions issued presented in Sec. III. Additionally, the influence of data size and cache usage can be examined with cache optimizations of the basic program versions and comparisons with established library implementations, such as in BLAS or LAPACK.

References

[1] Intel Corporation, "Intel Intrinsics Guide," Apr. 2018. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

[2] 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," Intel® Corporation, Tech. Rep., 2013. [Online]. Available: http://www.intel.com.br/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-closing-ninja-gap-paper.pdf

[3] J. M. Cebrián, L. Natvig, and J. C. 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*, Nov. 2012. doi: 10.1109/SC.Companion.2012.93 pp. 675–684.

[4] M. Lorenz, L. Wehmeyer, and T. Dräger, "Energy Aware Compilation for DSPs with SIMD Instructions," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, ser. LCTES/SCOPES '02. New York, NY, USA: ACM, 2002. doi: 10.1145/513829.513847. ISBN 978-1-58113-527-5 pp. 94–101.

[5] Intel Corporation, "Intel C++ Compiler 17.0 Developer Guide and Reference," 2018. [Online]. Available: https://software.intel.com/en-us/node/682974

[6] G. H. Golub and C. F. Van Loan, *Matrix computations*, 4th ed. Baltimore, Md.: Johns Hopkins University Pr., 2013. ISBN 978-1-4214-0794-4

[7] T. Jakobs, M. Hofmann, and G. Rünger, "Reducing the Power Consumption of Matrix Multiplications by Vectorization," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, Aug. 2016. doi: 10.1109/CSE-EUC-DCABES.2016.187 pp. 213–220.

[8] M. Plauth and A. Polze, "Are Low-Power SoCs Feasible for Heterogenous HPC Workloads?" in *Euro-Par 2016: Parallel Processing Workshops*, vol. 10104. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-58943-5_61. ISBN 978-3-319-58942-8 978-3-319-58943-5 pp. 763–774.

[9] T. Rauber and G. Rünger, "Towards an Energy Model for Modular Parallel Scientific Applications," in *2012 IEEE International Conference on Green Computing and Communications*, Nov. 2012. doi: 10.1109/GreenCom.2012.79 pp. 523–532.

[10] H. Lien, L. Natvig, A. A. Hasib, and J. C. Meyer, "Case Studies of Multi-core Energy Efficiency in Task Based Programs," in *ICT as Key Technology against Global Warming*. Springer, Berlin, Heidelberg, Sep. 2012. doi: 10.1007/978-3-642-32606-6_4 pp. 44–54.

[11] H. Caminal, D. Caballero, J. M. Cebrián, R. Ferrer, M. Casas, M. Moretó, X. Martorell, and M. Valero, "Performance and energy effects on task-based parallelized applications: User-directed versus manual vectorization," *The Journal of Supercomputing*, Mar. 2018. doi: 10.1007/s11227-018-2294-9

[12] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '14. New York, NY, USA: ACM, 2014. doi: 10.1145/2568058.2568068. ISBN 978-1-4503-2653-7 pp. 57–64.