NTNU – Trondheim
Norwegian University of
Science and Technology

Increasing SpMV Energy Efficiency Through Compression

A study of how format, input and platform properties affect the energy efficiency of Compressed Sparse eXtended

Lars-Ivar H Simonsen

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Lasse Natvig, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Assignment

Energy-efficiency of CSX on Intel Ivy Bridge processors and the Vilje Supercomputer

(Master thesis project text, proposal pr. 29/1-2013, LN.)

This is a continuation of the autumn project "Energy efficiency of CSX" by Lars-Ivar Hesselberg Simonsen. The project contributes to NTNUs participation in the PRACE project `http://www.prace-project.eu/`, and is also in cooperation with GRNET in Athens. The University of Tokyo is another potential collaboration partner for the project.

An overall goal of the master thesis project will be to study the energy efficiency of CSX in more depth by developing appropriate measurement techniques and/or tools that make it possible to examine the impact on energy-efficiency of

(a) structural properties of matrices (regularity, size)

(b) variable formats (functions of compression rate in CSX and at least one more "traditional" format such as CSR)

(c) degree of clock frequency, parallelism and single vs. multiple socket execution on a 2 x 8 core Sandy Bridge platform.

It is expected that the main execution platform is the CARD server using one or two sockets. It is considered part of the project to do similar tests on one node of the Vilje supercomputer and discuss differences - if there are any. These tests can be restricted to performance only if appropriate energy measurement techniques is not made available to the student during the first half of the master thesis project.

The master thesis should contain a short discussion presenting ideas on how CSX can be modified or extended to be used in an application spanning multiple nodes on Vilje and similar supercomputers.

If time permits, the project can be extended by augmenting the study with

(i) The effects of hyper-threading and Turbo Boost Technology.

(ii) Energy-efficiency studies on more execution platforms, such as a quad core Ivy Bridge desktop (minvilje.idi.ntnu.no) and a quad core Sandy Bridge desktop (festinalente.idi.tnu.no). This should then be accompanied by a discussion of performance and energy-efficiency differences between the different platforms.

Main supervisor: Professor Lasse Natvig (CARD-group, IDI). Technical co-supervisor: PhD Jan Christian Meyer (High Performance Computing Section, NTNU - IT Dept.).

Abstract

This work is a continuation and augmentation of previous energy studies of Compressed Sparse eXtended (CSX), a framework for efficiently executing Sparse Matrix-Vector Multiplication (SpMV).

CSX was developed by the CSLab at the National Technical University of Athens (NTUA), and utilizes compression to overcome a significant memory bottleneck inherent in SpMV, thus increasing performance and energy efficiency of its execution.

SpMV is notorious within scientific computing for its low performance. However, the problem is unavoidable, as SpMV can be found within several scientific applications. In this work, CSX is tested as the SpMV kernel in a framework implementing the Conjugate Gradient Method (CG), an iterative algorithm for solving specific linear algebra problems. CSX is also evaluated against Compressed Sparse Row (CSR), a storage scheme widely used when executing SpMV.

This work augments existing studies by evaluating properties in the formats themselves, in the matrices used as input and in the target platform to gain knowledge on how to maximize the benefits of CSX, as well as for what cases CSX does not prove beneficial. The work also compares the performance of SpMV-execution on a stand-alone server known as the CARD-server to similar execution on the Vilje supercomputer. This is done to evaluate how the differences between these two machines affect the results.

Based on the results, it is shown that CSX should be used for matrices larger than the Last Level Cache (LLC) of the target machine and for matrices with high degrees of clustering in their values. The best energy efficiency trade-offs are found at eight threads on dual socket configurations, and this is shown to be related to the amount of physical cores per CPU. Similarly, frequency throttling is shown to increase the energy efficiency of the execution only at high numbers of threads and at the cost of performance.

Overall, CSX is shown to obtain higher energy efficiency than CSR for SpMV-execution, given a suitable problem and run configuration. Thus, it is confirmed that CSX can be used to decrease the energy consumption of SpMV applications.

Sammendrag

Dette arbeidet er en fortsettelse og utvidelse av tidligere studier av Compressed Sparse eXtended (CSX), et rammeverk for effektiv utførelse av glissen matrisevektor multiplikasjon (Sparse Matrix-Vector Multiplication, SpMV).

CSX ble utviklet av computersystemlaboratoriet (CSLab) ved det Nasjonale Tekniske Universitet i Aten (NTUA), og utnytter kompresjon av data for å motvirke en vesentlig minneflaskehals i SpMV. Som sådan øker CSX ytelsen og energieffektiviteten til SpMV-eksekveringen.

SpMV er beryktet innen vitenskapelig beregning grunnet sin lave ytelse, men er uunngåelig grunnet sin plass i mange vitenskapelige applikasjoner. Dette arbeidet tester CSX som SpMV-kjernen i et rammeverk som implementerer konjugerte gradienters metode (Conjugate Gradient Method, CG), en iterativ løser av spesifikke problemer innen lineær algebra. CSX blir også testet opp mot Compressed Sparse Row (CSR), et lagringsformat for matriser som ofte blir brukt for eksekvering av SpMV.

Dette arbeidet utvider eksisterende studier ved å evaluere egenskaper i formatene, i matrisene og i platformene brukt for eksperimentene for å få innsikt i hvordan maksimere ytelses- og energiforbedringene til CSX, samt når CSX ikke bør benyttes. Arbeidet sammenligner i tillegg ytelsen til SpMV-eksekvering på en frittstående tjener kjent som CARD-serveren mot tilsvarende eksekvering på stormaskinen Vilje. Dette gjøres for å evaluere hvordan forskjeller mellom disse to maskinene påvirker resultatene.

Det blir vist, basert på resultatene, at CSX bør brukes for matriser som er større en siste-nivå-cachen (Last Level Cache, LLC) til maskinen som gjør eksekveringen, og for matriser med stor grad av gruppering blant verdiene. De beste energieffektivetsresultatene finnes ved kjøringer med åtte tråder på to sokler, og dette blir vist å være relatert til antall fysiske kjerner per CPU. Tilsvarende blir frekvensstruping vist å øke energieffektiviteten kun ved eksekveringer med høyt antall tråder, og da på bekostning av ytelse.

På generell basis blir det vist at CSX gir høyere energieffektivitet enn CSR, gitt et passende problem og eksekveringskonfigurasjon. Det blir dermed bekreftet at CSX kan brukes til å senke energiforbruket til SpMV-applikasjoner.

Acknowledgements

I would like to thank the following people for their help and input during the project:

- **Lasse Natvig** for supervising the project, providing feedback on the report and results, and offering guidance throughout the course of this work.
- **Jan Christian Meyer** for co-supervising the project, offering guidance in the technical aspects of the work and technical feedback on the results obtained.
- **Juan Cebrian** for providing help and administration with the CARD-server and the Yokogawa wall power meter.
- **Egil Holvik** for help with performing condition number estimation on the Kongull cluster.
- **Vasileios Karakasis** for help and input regarding the statistical sampling errors identified for CSX.

Contents

Contents i

List of Figures iii

List of Tables v

Glossary vii

1 Introduction 1
- 1.1 Project Purpose . . . 2
- 1.2 Motivation . . . 2
- 1.3 Structure of the Report . . . 3

2 Background 5
- 2.1 SpMV . . . 5
- 2.2 Conjugate Gradient Method . . . 6
  - 2.2.1 Conjugate Vectors . . . 6
  - 2.2.2 The Quadratic Form . . . 7
  - 2.2.3 The Conjugate Gradient Method . . . 8
  - 2.2.4 Convergence . . . 11
- 2.3 CSR . . . 12
- 2.4 CSX . . . 12
  - 2.4.1 Preprocessing . . . 14
    - Statistical sampling . . . 14
  - 2.4.2 Parallelization . . . 14
- 2.5 Energy Measurement . . . 15
  - 2.5.1 The RAPL interface in the MSR of Sandy Bridge . . . 16
  - 2.5.2 MSR Framework . . . 16
    - Continuous MSR data collection . . . 17
  - 2.5.3 Yokogawa WT210 . . . 17

3 Methodology 19
- 3.1 Hardware Setup . . . 19
- 3.2 Software Setup . . . 21

3.2.1 Software and versions 21
3.2.2 Software configuration and input 22
3.3 Matrices 25

4 Results 27
4.1 Format properties 28
4.1.1 Energy profile 28
CSR 29
CSX 30
CSX with statistical sampling 31
4.1.2 Comparison 33
CG execution 34
Full application energy 34
4.2 Matrix properties 37
4.2.1 Energy per nonzero 37
4.2.2 Energy consumption optimizability through CSX 42
4.3 Platform properties 46
4.3.1 Parallelization 46
4.3.2 Dual sockets 50
4.3.3 Clock frequency 53
4.4 Comparison of Vilje and the CARD-server 56

5 Discussion 59

6 Conclusion and Further Work 75
6.1 Conclusion 75
6.2 Further Work 77

Bibliography 79

Appendices 84

A Matrix Structures 85

List of Figures

2.1 Conjugate vectors [33] 7
2.2 Quadratic form plots [33] 8
2.3 An example of CSR for a sparse matrix [25] 12
2.4 An example of CSR-DU [25] 13
2.5 Yokogawa WT210 setup (Source: Intel [5]) 18

3.1 Vilje node setup (Source: HPC NTNU [1]) 20

4.1 Power-time plot for four threaded af_5_k101 CSR 29
4.2 Power-time plot for four threaded af_5_k101 CSX 31
4.3 Power-time plot for four threaded af_5_k101 CSX w/SP 32
4.4 Comparison of CG execution 35
4.5 Comparison of full application execution 36
4.6 Energy/nnz for all matrices 39
4.7 Energy/nnz for middle matrices 40
4.8 Energy/nnz for belt matrices 41
4.9 EDR for all matrices 43
4.10 Effect of parallelization on runtime 47
4.11 Effect of parallelization on energy 48
4.12 Average EDR for matrix groups 50
4.13 Energy difference between one and two sockets 51
4.14 Speedup of dual sockets 52
4.15 Average runtime for all matrices on differing frequencies 54
4.16 Average energy consumption for all matrices on differing frequencies 54
4.17 Average EDP = $E \times T^2$ for all matrices on differing frequencies . . . 56
4.18 Runtime of CARD and Vilje 57

5.1 L3 misses per nnz 64
5.2 L3 misses per second 66
5.3 L3 miss rates 67
5.4 Optimal configurations for the CG execution 69

A.1 Matrix Memory Structures (Source: UFlSMC [11]) 87

List of Tables

2.1 List of available RAPL sensors [22] 16

3.1 Vilje single node specification 20
3.2 MT_CONF core number configurations 22
3.3 XFORM_CONF configurations for CG binary 23
3.4 Additional options for CG binary 23
3.5 Environmental variables for statistical sampling 23
3.6 Variables for the test framework 24
3.7 Matrices used in experiments 26

4.1 Matrices that caused CSX with statistical sampling to terminate . . 33
4.2 low_nnz_group 38
4.3 middle_group 40
4.4 belt_group 41
4.5 csx_optimized_group 43
4.6 semi_optimized_group 44
4.7 csr_optimized_group 45

5.1 Required number of CG iterations 60
5.2 Average Pearson Product-Moment Correlation Coefficient 62
5.3 Pearson Product-Moment Correlation Coefficient for CARD and Vilje comparison 72

Glossary

BCSR Blocked CSR. 13

CARD Computer Architecture and Design. 2, 19, 21, 22, 24, 27, 56–58, 72, 76, 78

CG Conjugate Gradient Method. 2, 5–12, 14, 17, 22–30, 32–34, 37, 40, 42, 46, 50, 51, 57, 59–63, 65, 68, 71, 77

CPU Central Processing Unit. 15, 16, 19, 20, 22, 24, 27, 28, 30, 31, 46, 49, 50, 52–54, 56, 61, 63, 65, 68, 70–72, 76, 77

CSC Compressed Sparse Column. 12

CSR Compressed Sparse Row. 2, 5, 12, 13, 15, 22, 27–34, 37, 38, 40–42, 45, 46, 49–51, 53, 57, 60, 63, 65, 68, 70, 71, 76, 77

CSR-DU Compressed Sparse Row with Delta Units. 12, 13

CSX Compressed Sparse eXtended. 1, 2, 5, 6, 12–16, 22–34, 37, 38, 40–46, 49–51, 53, 56, 57, 60–63, 65, 68, 70–73, 75–78

DLP Data level parallelism. 6, 14, 29, 46, 49

EDP Energy Delay Product. 55, 68

EDR Energy Decrease Ratio. 42, 44, 46, 49, 50

GCC GNU Compiler Collection. 22, 58

GPU Graphics Processing Unit. 6, 78

HPC High-Performance Computing. 1, 2, 55

LLC Last Level Cache. 20, 25, 26, 42–45, 62, 63, 65, 68, 70, 71, 75–77

MPI Message Passing Interface. 72

MSR Model-specific register. 16, 17, 19, 21, 22, 28, 30, 31, 33, 34, 60, 61, 72, 78

nLSP non-homogeneous linear system problem. 6, 7, 12, 59

nnz non-zero. 12, 15, 24–26, 29, 32, 37, 38, 40–44, 49, 61–63, 72, 75

NTNU Norwegian University of Science and Technology. 2, 21

NTUA National Technical University of Athens. 1, 2, 77

PCC Pearson product-moment correlation coefficient. 61, 72

PRACE Partnership for Advanced Computing in Europe. 2

RAPL Running Average Power Limit. 15–17, 19, 21, 22, 28, 56

SMT Simultaneous multithreading. 22, 27, 34, 46

SpMV Sparse Matrix Vector Multiplication. 1, 2, 5, 6, 12–14, 28, 29, 46, 55, 63, 65, 71, 73, 75–77

Chapter 1

Introduction

Over the past few years, power and energy consumption has become the main limiting factor for computing in general. This can both be seen at the architectural level [19], and at the application level, especially within High-Performance Computing (HPC) and other compute heavy data center activities [30].

In order to overcome these problems, energy efficiency will have to be in focus both when designing architectures, and when creating applications to be run on them. This requires studies of existing and emerging technologies, in order to identify areas of potential energy consumption decrease, as well as studies on how to exploit them.

Within scientific HPC, one of the more frequent problems that that is still considered a bottleneck in many applications is Sparse Matrix Vector Multiplication (SpMV). This problem exhibits a large memory bottleneck and is thus notorious for limiting the performance of scientific applications. SpMV is found within several HPC problems, most notably as a bi-product when solving large linear systems.

Compressed Sparse eXtended (CSX) is one of many optimizations aimed at overcoming the memory bottleneck of SpMV. It does this by using compression to limit the size of the in-memory working set, thus limiting the bottleneck at the cost of additional processing. CSX was developed by the CSLab at National Technical University of Athens (NTUA), and is the focus of this work.

Previous studies, such as the work of Karaksis *et al.* [23], Simonsen [21] and Meyer *et al.* [28], have shown that the optimizations of CSX can increase the energy efficiency of SpMV applications for certain configurations. This work aims at exploring these configurations to gain knowledge on how, when and why CSX can save energy.

1.1 Project Purpose

The purpose of this project is to augment existing energy studies of CSX. This is done by expanding upon existing measurement techniques in order to gain knowledge on what cases make CSX save energy, and how to decide whether or not to use CSX for solving SpMV in the context of energy efficiency.

The goal of the project is to obtain knowledge about how differing properties in input and execution affect the energy consumption of CSX, as well as obtain viable parameters for its energy efficiency. Findings, especially potential parameters, are discussed to gain knowledge about how to maximize the energy efficiency, and potential trade-offs are explored.

In order to make the results relevant and realistic, CSX is tested as the SpMV-kernel in a framework implementing the Conjugate Gradient Method (CG). This is an iterative method for solving specific $Ax = b$ problems, often found within linear algebra. As this problem is frequently solved in the context of scientific HPC, any results are relevant to the energy consumption of HPC centers.

To evaluate the results of CSX, they are tested against a simpler, well tested format for solving SpMV. The format chosen for this is called Compressed Sparse Row (CSR), and has been used to solve SpMV for more than 15 years.

Lastly, the results obtained from executions on a dedicated server known as the CARD-server are compared to similar results on the Vilje supercomputer. This is done to evaluate the CARD-servers function as a testing platform for application to be run on Vilje with regards to their performance and energy efficiency.

1.2 Motivation

The main motivation behind this work is the increased focus on energy efficient computing seen within academic research over the past few years. This focus is a product of the increasing energy demands of computing reaching physical limitations.

This work is done in the hopes that the findings and potential parameters can be used to evaluate the input and target architecture in order to maximize the energy efficiency of SpMV applications. This, in turn, is done to help increase the energy efficiency of scientific HPC, by addressing the efficiency of one of its most frequently used kernels.

The work is a collaboration between the Computer Architecture and Design (CARD)-group at the Norwegian University of Science and Technology (NTNU) and the CSLab at the NTUA, the creators of the CSX library. Thus, it is a contribution to NTNUs role in the Partnership for Advanced Computing in Europe (PRACE), especially within the area of energy efficiency.

1.3 Structure of the Report

This Report is structured as follows:

Chapter 2 contains the necessary background information to understand further discussion. The Chapter has five major Sections.

Chapter 3 describes the machine, software and input setup used for the experiments. A description and discussion of the matrix set can also be found here. The Chapter has three major Sections.

Chapter 4 presents the results from the executions. This Chapters is split into four major Sections, each presenting different properties that are to be discussed.

Chapter 5 contains the discussions of the results in the previous Chapter. These discussions are made to relate findings to the Project Purpose, as well as discussions needed to interpret the results. This Chapter has one major Section.

Chapter 6 contains the conclusions that can be drawn based on the results and discussions, as well as thoughts on things that should be explored in a continuation of this work. This Chapter has two major Sections.

Chapter 2

Background

This Chapter contains background information needed to fully understand the contents of this work. The Chapter is split into three major parts:

1. An introduction to SpMV and CG. These are the problem solved by CSX, and a problem bound by SpMV used to evaluate the performance of CSX, respectively.
2. An introduction to CSR and CSX, showing the conceptual idea behind them, how they optimize the execution of SpMV and how they are different.
3. A description of the energy measurement techniques used in this work, as well as a discussion of what techniques should be used and why.

2.1 SpMV

Sparse Matrix Vector Multiplication is the operation of multiplying a sparse matrix A with a dense vector x (Ax). It is conceptually the same as as dense matrix-vector multiplication, however, the sparsity of the matrix highly affects the performance of the multiplication. As was pointed out by Williams *et al.*:

> SpMV is a frequent bottleneck in scientific computing applications, and is notorious for sustaining low fractions of peak performance. [36]

This is due to the memory access pattern of the matrix, which pushes SpMV from the computationally bound realm of dense matrix-vector multiplication into a memory bound realm. Because of this, SpMV has gained a lot of interest from computer scientists, resulting in several methods of optimization. Some of these optimizations include:

The CSR memory format, which is an optimization used as early as 1994 in the work of Saad [31], OSKI, which is a collection of automatically tuned SpMV kernels

presented by Vuduc *et al.* in 2005 [35] and lately CSX, which is a compression based optimization for SpMV presented by Kourtis *et al.* in 2011 [25].

Like dense matrix-vector multiplication, SpMV is highly parallelizable due to the high degree of Data level parallelism (DLP). This makes the use of parallel computing an easy and effective way of optimizing the SpMV runtime. This has been shown in the work of Williams *et al.* [36], who compared the effect of several multicore optimizations to earlier SpMV optimizations, and later in the work of Bell *et al.* [14], who implemented several SpMV kernels on a highly parallel Graphics Processing Unit (GPU).

SpMV is commonly used in the problem of solving $Ax = b$ for a non-zero vector x, where A is a sparse $n \times n$ matrix and b is a known vector. This problem, hereafter called the non-homogeneous linear system problem (nLSP), is a common and important problem within linear algebra and scientific computing. For this reason, it will be used to study CSX's performance.

2.2 Conjugate Gradient Method

In order to gain insight into the performance and energy efficiency of CSX, a benchmark is required. As was noted earlier, SpMV is commonly found in linear algebra problems. Hence, a benchmark based on nLSP will provide a realistic workload for SpMV.

The Conjugate Gradient Method is an iterative algorithm for solving symmetric, positive-definite nLSP. As this implies iteratively applying SpMV to the target matrix, its performance is bound by the performance of the SpMV solver. This makes it well suited as a realistic performance benchmark for CSX and other SpMV solvers.

This Section provides a conceptual explanation of the CG algorithm, and hence, the background information required to discuss the performance of CG. The Section is based on the work of Shewchuk [33], which provides an in-depth study of the topic.

First, two principles are required in order to explain CG: *Conjugate Vectors* and *the Quadratic Form*.

2.2.1 Conjugate Vectors

Two vectors are said to be *conjugate* if they are *orthogonal* to each other with respect to a matrix A (also known as *A-orthogonal*). For the two vectors v and u, this is denoted as

$$v^T Au = 0$$

An example of such vectors is shown in Figure 2.1. The vector-pairs in Figure 2.1(a) are conjugate because the vector-pairs in Figure 2.1(b) are orthogonal.

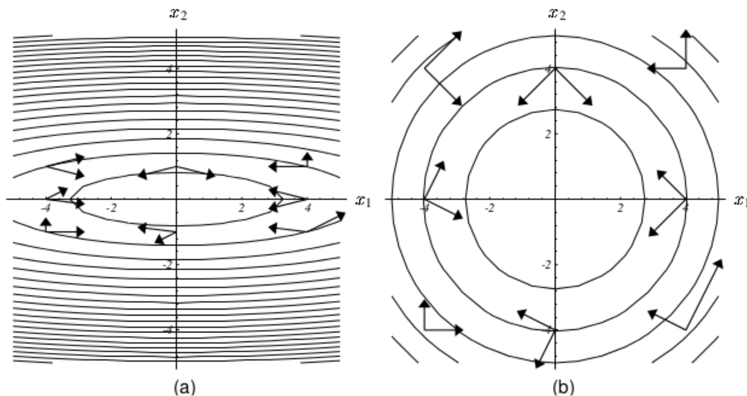


Figure 2.1: Conjugate vectors [33]

Another property of conjugate vectors was presented in a Lemma in the work of Chong *et al.* [18]:

> **Lemma 10.1** Let Q be a symmetric positive definite $n \times n$ matrix. If the directions $d_0, d_i, ..., d_k \in \mathbb{R}^n, k \leq n-1$, are nonzero and Q-conjugate, then they are linearly independent. [18]

This property of conjugate vectors can be related to solving nLSP in the following manner: A set of n mutually conjugate vectors $\{d_i\}$ will, because of Lemma 10.1, form a basis for $\mathbb{R}^n$. If we then look at nLSP

$$Ax = b \tag{2.1}$$

and the fact that a vector (x) can be created from a basis of $\mathbb{R}^n$

$$x = \sum_{i=1}^{n} \alpha_i d_i \tag{2.2}$$

we see that nLSP can be presented as

$$b = \sum_{i=1}^{n} \alpha_i A d_i \tag{2.3}$$

This implies that the solution of $Ax = b$ can be found by iteratively solving Equation 2.3 for the n mutually conjugate vectors of the set $\{d_i\}$. This is the general principle behind CG.

2.2.2 The Quadratic Form

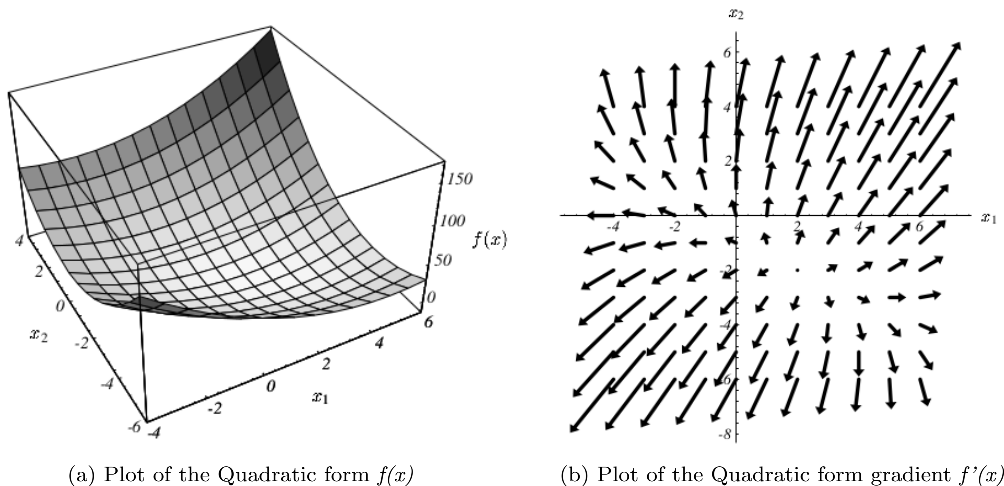
The quadratic form is defined as a scalar function given as

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{2.4}$$

It can be shown (consult Shewchuk [33] Section 3 for details), that this function is minimized for the solution of $Ax = b$ if A is a symmetric ($A = A^T$), positive-definite ($v^T Av > 0$) matrix. This implies that the gradient of the function $f(x)$ is given by the following equation.

$$f'(x) = Ax - b \tag{2.5}$$

To illustrate this property, an example plot of $f(x)$ is shown in Figure 2.2(a). In this plot, the solution of $Ax = b$ is found at $x = [2, -2]^T$.



(a) Plot of the Quadratic form *f(x)*

(b) Plot of the Quadratic form gradient *f'(x)*

Figure 2.2: Quadratic form plots [33]

The gradient of the plot in Figure 2.2(a) is shown in Figure 2.2(b). As we see from this plot, the gradient for each of the scalar values of $f(x)$ points in the direction of the steepest increase in $f(x)$. Intuitively, this is approximately opposite of the direction of x, and can be used in order to quickly converge to x, as will be shown in the following Section.

2.2.3 The Conjugate Gradient Method

Having defined Conjugate Vectors and the Quadratic Form, the algorithm of CG can be explained. This Section starts by introducing the basic steps of the CG algorithm, followed by a more in depth explanation of each of the steps. The reader is referred to Shewchuk [33] Section 8 for detailed explanations and proofs.

Listing 2.1 contains pseudo code for the CG algorithm.

```
1 Select a starting vector x at random.
2 Select a starting direction for the algorithm.
3 n times do:
4     Pick a step size to move along the selected direction.
5     Perform the step, updating x.
6     Calculate a new direction to move, conjugate to all
          previous directions.
```

Listing 2.1: Pseudo code for CG

As can be readily seen from this pseudo code, the algorithm iterates n conjugate directions in order to converge on the solution. This was in Section 2.2.1 (Equation 2.3) shown to produce the solution vector as long as each step is of the appropriate length. The problem, therefore, divides into two subproblems: finding the appropriate step length and finding conjugate directions.

In order to solve these problems, the CG algorithm introduces a new vector r. This vector, known as the residual vector, is calculated based on the quadratic form gradient defined in Section 2.2.2 (Equation 2.5). As was shown, the quadratic form gradient in each point of the quadratic form field points in the direction of the steepest increase. Thus, we know that the negation of this vector points approximately towards the solution of the system. This gives us step 2 in Listing 2.1:

$$d_0 = r_0 = -f'(x_0) = b - Ax_0 \tag{2.6}$$

From this step, we set both the initial direction d and the initial residual vector r to the negation of the quadratic form gradient. This is done to ensure quick convergence on the solution.

To find the length of each step (step 4 in Listing 2.1), we need to calculate the α_i of Equation 2.3. This is done through the following Equation:

$$a_i = \frac{r_i^T r_i}{d_i^T A d_i} \tag{2.7}$$

The idea behind this Equation is to ensure that the remaining error is orthogonal to the direction chosen, as this implies that the step in that direction moves the temporary x vector to the point of the solution vector along the chosen direction. This, in turn, means that the direction will not have to be traversed again when calculating the solution vector. The derivation of Equation 2.7 can be found in Shewchuk [33] Equations (30), (32), (42) and (46).

Once the direction and step length has been found, the actual step can be performed (step 5 in Listing 2.1):

$$x_{i+1} = x_i + \alpha_i d_i \tag{2.8}$$

This leaves one problem: finding directions conjugate to all previous directions traversed. We start by calculating a new residual vector. However, instead of

using the negated quadratic form gradient ($r_i = b - Ax_i$), we perform the following optimization:

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i d_i \\ -Ax_{i+1} &= -Ax_i - \alpha_i A d_i \\ b - Ax_{i+1} &= b - Ax_i - \alpha_i A d_i \\ r_{i+1} &= r_i - \alpha_i A d_i \end{aligned} \tag{2.9}$$

This is done to avoid performing two costly matrix-vector products, as Ad_i is already performed by Equation 2.7. The results can hence be reused when calculating the new residual vector.

This new residual vector is used to create a new search direction. In the discussion of Equation 2.7, it was stated that the remaining error (i.e. the residual vector) is orthogonal to the current search direction, and thus orthogonal to all previous search directions. This fact, along with a process called *the Gram-Schmidt conjugation* is used to find a new conjugate search direction.

The Gram-Schmidt conjugation is used in order to generate a set of n conjugate directions $\{d_i\}$ based on n linearly independent vectors $\{u_i\}$. Its formula is stated in Equation 2.10 and 2.11.

$$\begin{aligned} d_0 &= u_0 \\ d_i &= u_i + \sum_{k=0}^{i-1} \beta_{ik} d_k \end{aligned} \tag{2.10}$$

where β_{ik} is defined as

$$\beta_{ik} = -\frac{u_i^T A d_k}{d_k^T A d_k} \tag{2.11}$$

This process constructs the new direction d_i by taking the new linearly independent vector u_i, and subtracting the non-conjugate components of the previous directions. For the details of this, the reader is referred to Shewchuk [33] Section 7.2.

For CG, however, the process can be simplified. r_i has been shown to be orthogonal to all previous search directions $\{d_0, ..., d_{i-1}\}$. However, when constructing new search directions by performing Gram-Schmidt from residual vectors ($u_i = r_i$), r_i becomes orthogonal to all previous residual vectors as well $\{r_0, ..., r_{i-1}\}$. This, along with the fact that Equation 2.9 states that each new residual vector is formed from a linear combination of the previous residual and Ad_{i-1}, can be shown (see Shewchuk [33] Section 8) to imply that r_{i+1} is conjugate to all previous search directions, except d_i.

This means that Equation 2.10 can be simplified to the following:

$$d_{i+1} = r_{i+1} + \beta_{i+1} d_i \tag{2.12}$$

where β_{i+1} is defined as:

$$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \tag{2.13}$$

This solves the last of the subproblems, and hence, completes the last step (step 6 in Listing 2.1) of the algorithm.

For convenience, Listing 2.2 shows an implementation of the full algorithm, done by Jan Christian Meyer.

```
1  #! /usr/bin/octave
2
3  # Initialize A, b, x
4  # Compute the first residual (and the first direction)
5  d = r = b - A*x
6
7  for i=1:n
8      rdot = r' * r;   # Pick step size, based on residual
9      alpha = rdot / dot(A*d,d);
10     x = x + alpha * d;   # Step in that direction
11
12     r = r - alpha * A * d;   # Compute new residual
13     beta = r' * r / rdot;    # Gram-Schmidt constant
14     d = r + beta * d;    # New direction
15 end
```

Listing 2.2: Implementation of CG

As we can see from this Listing, the steps roughly relate to Equations (2.6), (2.7), (2.8), (2.9), (2.13) and (2.12).

2.2.4 Convergence

From the previous Subsections, we see that CG converges in n iterations, as this means traversing all n mutually conjugate directions. However, all n iterations are not always necessary.

It can be shown (Shewchuk [33] Section 9 and Saad [32] Section 6.11.3) that the error at iteration m is given by the following equation

$$||x - x_m||_A \leq 2 \left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^m ||x - x_0||_A \tag{2.14}$$

Where $||x||_A$ denotes the norm as $||x||_A = (Ax, x)^{\frac{1}{2}}$ [32] and κ is known as the condition number of the matrix.

What this formula shows is that based on the condition number of the matrix, which is defined as $\kappa(A) = \left|\frac{\lambda_{max}(A)}{\lambda_{min}(A)}\right|$ where $\lambda(A)$ denotes the Eigenvalues of a matrix A, the algorithm will quickly converge on the answer. Because of this, one might reach a suitable solution to the problem without having to do all n iterations. However, this is dependent on the matrix in question, as its Eigenvalues decides

the condition number of the matrix. The greater the condition number, the slower the algorithm will converge, and thus more iterations will have to be performed.

2.3 CSR

The definition of sparse matrices is that they consist mostly of zeros. Because of this, it is inefficient to store entire sparse matrices in memory. This gives rise to the obvious optimization of storing only the non-zero (*nnz*) values of the matrix, as it eliminates most of the matrix without reducing the amount of information.

This is the reasoning behind CSR, which is a storage scheme for storing only the *nnz* values of sparse matrices. Figure 2.3 shows an example of how this is accomplished. In short, CSR stores the matrix as three arrays, consisting of *row_ptr* and *col_ind*, which give the location of an element, as well as *values*, which contains the actual *nnz* value. Another possible approach would be Compressed Sparse Column (CSC), which is the same technique, only column-major [25].

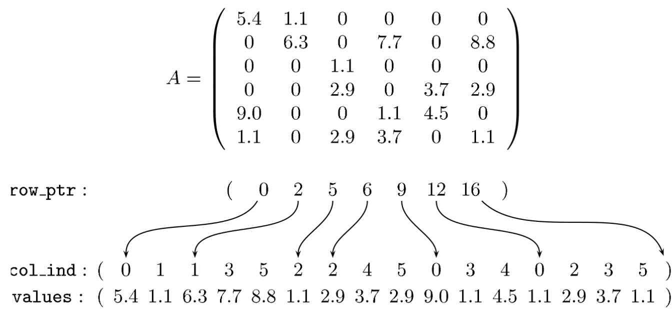


Figure 2.3: An example of CSR for a sparse matrix [25]

As this shows, CSR is not an algorithm for solving SpMV, but rather a storage optimization. As such, it is often used in conjunction with regular SpMV kernels in CG in order to solve nLSP effectively. CSR provides a significant memory and bus improvement, increasing the SpMV performance and thus increasing the performance of CG.

2.4 CSX

CSX, as opposed to CSR, is an algorithm designed for optimizing SpMV for multicore environments. It is based on the Compressed Sparse Row with Delta Units (CSR-DU) storing scheme for sparse matrices, but utilizes extended compression as well as code generation in order to further optimize memory access patterns and reduce the memory bottleneck.

In this Section an overview of how CSX operates will be presented. A more in depth description can be found in the presentation of CSX by Kourtis *et al.* [25].

As we have seen, the CSR optimization decreases the memory requirements of SpMV significantly, but there is still potential for improvement. There are several methods to achieve this, including Blocked CSR (BCSR) and CSR-DU [25]. The latter, presented by Kourtis *et al.* in 2008 [24], is a compression scheme based on delta encoding for CSR. It works by searching the *col_ind* array for groups (known as *units*) of values and representing them as delta values to a number representing the group. This substantially reduces the strain on the memory system, as the compression can reduce the data size significantly, depending on the matrix. An example can be seen in Figure 2.4.

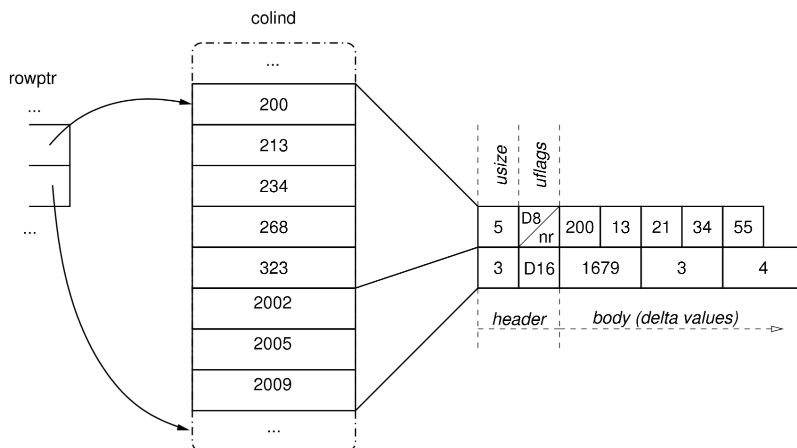


Figure 2.4: An example of CSR-DU [25]

CSX is built on the same principles as CSR-DU. It uses the same delta compression, but can detect groups in not only horizontal patterns (like CSR-DU), but also vertical, diagonal, anti-diagonal and 2D patterns. This is done by traversing the matrix in the desired directions and detecting groups along the way. For 2D patterns, a simple transformation from 2D to 1D space is used. This is implemented by splitting the matrix into bands (multiple rows/columns) and then transforming them to a linear access pattern. Once transformed, the simple 1D group detector can be used. When the detection is completed, the resulting 1D pattern is transformed back to 2D with the reverse transform.

When the entire matrix has been traversed in the different directions and the new matrix containing groups of delta values has been constructed, the algorithm generates specialized code for the SpMV operation kernel. This gives optimized access patterns based on the matrix, and hence further optimizing the memory use and runtime of the application.

2.4.1 Preprocessing

While there is a significant runtime improvement from using CSX, it comes at the cost of the preprocessing required to analyze the matrix. This can be viewed as a trade-off, where one should only use CSX for applications where the execution time is the dominating part of the runtime, e.g. problems that require several CG iterations. The complexity of the preprocessing is found to be *O(nnz)* [25].

As was stated above, the preprocessing of CSX is designed to find linear, diagonal and 2D patterns in the matrices. Because of this, the layout of the matrices will greatly affect the performance of CSX, where matrices with patterns identifiable by the preprocessing (e.g. diagonal matrices), will benefit significantly more from the optimizations than matrices with more irregular patterns (e.g. curved matrices).

The preprocessing itself is split into three phases. In the first phase, the matrix is searched for suitable patterns to encode using delta compression. The second phase is the actual matrix encoding. In this phase, the matrix is encoded with the patterns identified in the previous phase and stored in memory. The third phase is the code generation, where the SpMV kernel is generated based on the patterns used to encode the matrix.

Statistical sampling

In order to minimize the cost of the preprocessing, CSX offers a technique called statistical sampling. This approach reduces the work of the *unit* identification in the preprocessing significantly by reducing the amount of the matrix that is searched. Normally, the entire matrix is searched for all patterns while keeping a statistic for the best matching pattern in the different parts of the matrix. With statistical sampling enabled, the pattern identification is limited to a subset of the matrix based on a portion variable. This significantly limits the amount of work required by the preprocessing, thus reducing its runtime.

On the other hand, the use of this statistical sampling limits number of patterns identified in the matrix. This is because the pattern search space is severely limited, resulting in suboptimal patterns possibly being chosen. As a result of this, the use of statistical sampling will most likely result in reduced performance during SpMV execution. Hence, one can consider the use of statistical sampling a trade-off between reduced preprocessing cost and SpMV performance gain.

2.4.2 Parallelization

In order to achieve performance, CSX utilizes parallel execution on the parts of execution with high DLP. This is primarily two parts of the execution, namely the preprocessing and the SpMV execution. Both of these phases consist of matrix evaluation and calculation, making them well suited for multithreaded execution.

The parallelization done by CSX consists of splitting the *nnz*s of the matrix into groups based on the number of threads. This is done by iteratively dividing the remaining *nnz* count by the number of threads not yet initialized in order to create a limit value. Then, rows starting from the last processed row are iteratively added to the current matrix subset as long as the limit value for *nnz*s is not exceeded. This is done to ensure equal distribution of work for the threads, despite the variable row length offered by the CSR format. Upon reaching the limit value, the matrix subset is assigned to the current thread, the remaining *nnz* count is updated and the iteration continues to the next thread.

Upon successfully splitting the matrix, the now multithreaded execution can commence. Due to the equal distribution of *nnz*s, the execution of the threads is ensured to be of equal length when assuming no contention.

2.5 Energy Measurement

The goal of this work is to analyze and evaluate the energy consumption of CSX. This requires techniques for measuring energy at runtime, which can be done in various ways. The following Section describes the techniques used to achieve this.

Energy measurement has been the goal of much academic work, and hence, many different techniques have been used. Hähnel *et al.* evaluated the then relatively new Running Average Power Limit (RAPL) interface, which is a performance counter for energy made available through a register on the Central Processing Unit (CPU). Their work showed accurate results for the interface when compared to an external measurement tool [22]. Other measuring techniques include the use of wall power meters. This approach is suggested by Feng *et al.* in their description of Green500 [20], which is a energy efficiency top list for supercomputers. However, as they point out in their work, there are many ways to connect the power meter to the measured computer.

Finally, there has been a lot of research into power estimation using performance counters. This approach has been used by, among others, Bircher *et al.*, who provided a simple 2-input model based on performance counters [16], Singh *et al.*, who created a piece-wise function for power based on four performance counters [34] and Bertran *et al.*, who created a decomposable model for power for different parts of the CPU [15]. What is common to all these works is that they utilize performance counters that correlate well with the power consumption of the CPU to make a model for the power consumption.

For this work, the first approach of using the RAPL interface will primarily be utilized. This is done for three reasons:

1. The RAPL interface was shown by Hähnel *et al.* to provide sufficient accuracy in its measurements [22].

2. A framework for measuring the energy consumption of a system was developed by Lien for his Master Thesis [27], making measurement of the energy easier. This framework was also used in earlier research into the energy efficiency of CSX [21].
3. In this earlier work [21], it was shown that the energy consumption measured through the RAPL interface and energy consumption measured by a wall meter largely correlated. The only difference found was that the wall power meter more closely correlated with the runtime of the application. Hence, results from the RAPL interface should provide sufficient insight into both the CPU and full system energy consumption.

A wall power meter will also be used where appropriate to enhance the results from the Model-specific register (MSR) framework.

The rest of this Section contains a description of the measurement equipment and software used for the experiments. As this is the same equipment as was used in earlier research [21], the coming Sections are based on equivalent Sections in that report.

2.5.1 The RAPL interface in the MSR of Sandy Bridge

Intel was the first company to expose energy data from their processors [22]. This was done by adding an interface called the RAPL interface to the MSR of the Sandy Bridge microarchitecture processors. The interface consists of four registers, shown in Table 2.1, which can be read at runtime.

RAPL_PGK	Whole CPU package
RAPL_PP0	Processor cores only
RAPL_PP1	"A specific device in the uncore" [1]
RAPL_DRAM	Memory controller

Table 2.1: List of available RAPL sensors [22]

Values from these registers can be read each time the MSR is updated. Hähnel *et al.* found this to happen each 1 ms (or 2,500,000 cycle on a 2,5GHz), with jitter of about +/- 50,000 cycles [22]. They also found the registers to be sufficiently accurate when it comes to consumed energy, compared with a wall power meter.

2.5.2 MSR Framework

In order to measure energy consumed by the system while the experiments are running, one has to collect data from the MSR continuously as well as store this data. For this purpose, an energy measurement framework is used.

[1] GPU on Sandy Bridge and Ivy Bridge processors, not used in this document

The energy data collection framework used in this report was created by Hallgeir Lien for his Master Thesis on energy consumption in Sandy Bridge processors [27]. This framework is composed of two main parts, an energy library written in C that interacts with the RAPL interface, and a collection of Python scripts that allows users to create input files in which several parameters and environmental variables can be specified. Based on these input files, the framework sets up the environment and runs the specified executables a specified number of times while collecting data. At the end of each run, the total energy consumed is reported.

When all runs have been completed, the outputs are processed statistically based on the number of runs (i.e. mean, median and standard deviation is calculated), and all results are stored in an SQLite database.

The energy library also contains an interface written in C that allows for start and stop markers to be placed in the measured application code. Based on these markers, the library only measures energy for the code specified, such as the CG execution, instead of the entire application.

Continuous MSR data collection

The energy library is able to collect the overall energy consumption between two markers. However, in order to obtain continuous power readings from the MSR, several readings based on a set interval is required.

This was achieved with a small framework written in C by Jan Christian Meyer, that sends a recurring signal based on a configurable interval, and reads the MSR using the energy library upon catching this signal. The data collected is written to a buffer and written to file at the end of the application execution.

2.5.3 Yokogawa WT210

As was mentioned above, a wall power meter is used when appropriate to enhance the results from the MSR. The power meter in question is the Yokogawa WT210, which is used to measure the energy consumption of the entire system. This meter is connected between the power outlet and the computer being measured, and logs the power used by the system to a second computer, as illustrated in Figure 2.5. The second computer (named ESRV System in the Figure) is used in order to prevent instrumentation from affecting the energy consumption measurements of the target computer.

The Yokogawa WT210 is connected via serial interface to the ESRV System. On this system, a process called Intel Energy Server, part of the Intel Energy Checker SDK [4], retrieves power and energy information from the power meter. The collected data is written to a file specified in the process, and can be further aggregated in order to derive a desired output.

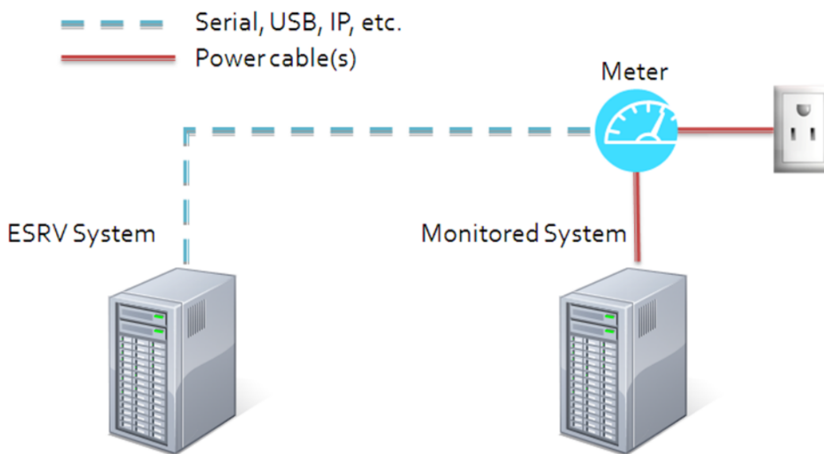


Figure 2.5: Yokogawa WT210 setup (Source: Intel [5])

Chapter 3

Methodology

In this Chapter, the test setup and input is presented. The first Section contains a description the hardware used for the experiments, both the CARD-server and the Vilje supercomputer. The second Section contains a description of the software used as well as its input parameters, and the last Section contains a presentation of the matrix set used in the experiments.

3.1 Hardware Setup

For the experiments in this report, two machines have been used: the supercomputer Vilje, and a dedicated server named the CARD-server. The experiments have primarily been run on the CARD-server, and thus, all results should be assumed to be based on the CARD-server unless explicitly stated otherwise.

As the hardware of the two machines are similar, however, a presentation of a single node on Vilje will be used to describe the overall hardware, followed by a discussion of the differences between the two machines.

The Vilje supercomputer is a 1404 node machine purchased by the Norwegian University of Science and Technology and met.no [7] in 2012. With its 22464 cores and theoretical peak performance of 467 Teraflop/s, it was ranked #44 in the Top500-list of June 2012 [13].

The specifications of a single Vilje node are presented in Table 3.1, and shown in Figure 3.1.

The most important feature of the specification are the Xeon E5-2670 processors. These processors are based on the Sandy Bridge microarchitecture, which is the first architecture to include the RAPL interface in the MSR. This allows for applications to read the energy consumption of the CPU and thus gain knowledge about the

CPU	2x Intel Xeon E5-2670 @ 2.6GHz
Cores	16 cores (8 per CPU) + 16 hyper-threads
L1 cache	32 kB data + 32 kB instruction per core, 8-way associative
L2 cache	256 kB per core, 8-way associative
L3 cache	20480 kB per physical CPU, 20-way associative
RAM	32GB

Table 3.1: Vilje single node specification

Figure 3.1: Vilje node setup (Source: HPC NTNU [1])

power consumption at runtime, which in turn allows for in depth studies of energy consumption of applications.

Besides the CPU microarchitecture, one should note the large Last Level Cache (LLC). Its size of 20MB allows for a large working data set near the processing cores, thus limiting the memory accesses and removing latency. This could potentially have a great impact on the performance of an application, depending on the input

size.

As was stated in the Assignment, access to power measurement on the Vilje supercomputer was not available at the start of this work. This access has not been made available during the course of this work, and thus, no energy results will be presented for Vilje. However, performance analysis and its impact on the energy efficiency of the applications run will be discussed in Chapter 4.

Moving on to the CARD-server, its overall hardware closely resembles that of a single compute node on Vilje. This server was built by the CARD group at NTNU with the purpose of representing the performance of a single Vilje node. This was done to allow for experimentation and performance estimation without having to allocate time on Vilje. As was stated in the beginning of this Section, this server was used for most of the experiments.

Specifications-wise, the CARD-server is computationally identical to a node on Vilje, with one exception: the CARD-server has *64 GB* of RAM, which is double that of Vilje. However, this should not affect the performance or energy consumption notably for problem sets with less than 32 GB inputs.

The machines themselves, on the other hand, are significantly different. A single node on Vilje is a rack-mounted compute node with no hard disks or similar equipment. This can be found in dedicated racks. The CARD-server, on the other hand, is a single node server with all the equipment this requires. This could potentially affect the power consumption measured by the wall power meter, but should not affect the power consumption measured by the MSR.

For the CARD-server, access to the MSR was given at the start of the work, and thus, energy results can be gathered from runs on this machine. In addition to RAPL counter access, this machine was fitted with a wall power meter, as described in Section 2.5.3. This allows for wall power to be measured, making comparisons between the RAPL counters and the actual consumed energy of the machine possible.

3.2 Software Setup

This Section describes the software used to run the experiments. The following information is split into two parts: first the software used and their versions are presented, followed by a description of the test setup and input.

3.2.1 Software and versions

Most of the tests in this work were run on the CARD-server, which is running Fedora release 17 (Beefy Miracle) built on GNU/Linux kernel version 3.5.2-3.fc17.x86_64. The Vilje supercomputer, on the other hand, is running SUSE Linux Enterprise Server 11 built on GNU/Linux kernel version 2.6.32.59-0.3-default.

The CSX software was built from a checkout of the CSX GitHub repository [3] done 2013-01-16. This is based on the 0.2 version of CSX, with a patch for NUMA released 2012-09-14. The CSX software was compiled with LLVM version 2.9 using the Clang frontend version 2.9.

All other compiled software, including LLVM, was compiled using the GNU Compiler Collection (GCC) version 4.6.3 for the CARD-server and 4.6.2 for Vilje.

For controlling the CPU frequency and setting power governor, which is the frequency scaling strategy of the CPU, cpufrequtils version 008, and later cpupower version 3.8.4-102 were used. The change in software was caused by security upgrades on the CARD-server.

The RAPL interface was read using a framework developed by Hallgeir Lien, discussed earlier in Section 2.5.2. All other performance counters from the MSR were read using PAPI version 5.1.0.

3.2.2 Software configuration and input

All the actual experiments have been run using an implementation of CG (see Section 2.2), provided by the CSX framework. This application supports solving CG for either CSX or CSR with configuration options for parallelization and degree of CSX compression.

MT_CONF core numbers	
0-7	Physical cores on CPU0
8-15	Physical cores on CPU1
16-23	SMT cores on CPU0
24-31	SMT cores on CPU1

Table 3.2: MT_CONF core number configurations

The MT_CONF variable is an environmental variable read by the CG application on startup. This variable dictates what cores (including Simultaneous multithreading (SMT)-cores) should be used to run the application. Hence, this variable dictates the parallelization of the application. The core numbers are given by the Linux subsystem and can be displayed with the commands *cpufreq-info* or *cpupower -c all frequency-info*.

Table 3.2 displays the core numbers set in the MT_CONF variable to set up thread configurations. Note that these core numbers are equal to the ones shown in Figure 3.1. In order to set specific configuration, the MT_CONF variable is set to the core number on which threads are to be run. For example, four threads running on four physical cores on CPU0 would have an MT_CONF of *0,1,2,3*. Four threads running on two physical cores with SMT on CPU0 would have an MT_CONF of *0,1,16,17*, while four threads running on four physical cores on both CPU0 and CPU1 would have an MT_CONF of *0,1,8,9*.

XFORM_CONF	
0	For CSR
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22	For CSX

Table 3.3: XFORM_CONF configurations for CG binary

The XFORM_CONF variable is another environmental variable read by the CG application on startup. This variable configures which patterns the CSX preprocessing searches for, and thus, which patterns can be detected and optimized by CSX. This variable is set to a series of numbers, each representing a different pattern for recognition. Refer to the CSX README on GitHub [3] for a detailed list of possibilities.

Unless stated otherwise, the configurations shown in Table 3.3 are used in the experiments. This means full compression for all configurations running CSX.

Additional CG options	
-l 1024	Number of CG-iterations
-L 1	Number of repetitions done by the binary
-x (Enabled or disabled)	Used with XFORM_CONF to enable CSX

Table 3.4: Additional options for CG binary

In addition to the environmental variables, the CG application is configurable through arguments passed to the application. The arguments set for the experiments in this work can be found in Table 3.4. For a full list of available arguments, refer to the README mentioned above.

The number of CG iterations is set to 1024 to ensure a sufficient runtime of the application to minimize energy anomalies in the measurements, while still keeping the runtime of the experiments at a manageable level. For real executions of CG, however, the required number of iterations is closer to the rank of the matrix (as discussed in Section 2.2). The number of repetitions is set to one, as repetition functionality is already provided by the energy measurement framework.

Statistical sampling	
SAMPLES	64/#threads
WINDOW_SIZE	N/A
SAMPLING_PORTION	0.01

Table 3.5: Environmental variables for statistical sampling

The statistical sampling, discussed in Section 2.4.1, is controlled through three environmental variables displayed in Table 3.5. The SAMPLES variable controls how many sample windows each thread of the application should process. Hence, in order to get equal statistical sampling regardless of the number of threads, it was

set based on the number of threads. The WINDOW_SIZE variable determines the size of each of the sample windows. This variable was not used in the experiments, as its value is trumped by the last variable. The SAMPLING_PORTION variable decides the amount of *nnz*s that are sampled during the preprocessing. This value then decides the sample window size based in Equation 3.1, rendering the value of the WINDOW_SIZE variable ignored.

$$SAMPLING_PORTION \times non_zeros = SAMPLES \times computed_win_size \tag{3.1}$$

The values of these variables were chosen to match the energy consumption study of Karaksis *et al.* [23]. The only difference is in the SAMPLES variable, where this value was set to 48/#threads due to the six-core architecture of their CPUs. The value of this variable was therefore increased to 64/#threads to accommodate the eight-core CPUs of Vilje and the CARD-server.

The last set of input variables are input for the test framework discussed in Section 2.5.2. These variables and their values are shown in Table 3.6. Note that the values shown in this Table are only the default values chosen. Unless stated otherwise, these are the values used.

Test framework		
reps	Number of repetitions per run	5
cpufreq	Frequency (MHz) at which to run test	2600 or N/A

Table 3.6: Variables for the test framework

As shown in the Table above, all the cores were clocked at 2,6GHz, i.e. maximum frequency, for the experiments. This was done by setting the power governor to *userspace* with *cpufrequtils* or *cpupower*, which allows users to set the clock frequency of the cores manually. The reason for this choice of frequency was to ensure maximum performance while keeping the results comparable. However, this is only applicable for the CARD-server.

For Vilje, on the other hand, no kernel cpufreq driver is loaded. This means that cpufrequtils is not allowed to monitor or alter the frequency or power governor of the cores, and thus, no specific frequency could be enforced from userspace. Calls to the Linux kernel confirm that the nodes are running at 2,6GHz, matching the default value for the experiments on the CARD-server.

When using PAPI with applications running POSIX threads, such as the CG application provided by the CSX framework, the event counts are counted on a per thread basis. As is described in the PAPI documentation [12], and made evident by a discussion on the PAPI mailing lists [9], PAPI counters started in the main thread do not include those of its children when using POSIX threads. As the inclusion of PAPI counters for each thread would require significant additions to the

CSX framework, it was not done for this work. However, the main thread of the CG application executes the same amount of work as every other thread. Because of this, an estimate of the total number of events can be made by multiplying the results for the main thread by the number of threads.

Hence, all results for multithreaded performance counters in this work, such as cache misses, are derived from this estimate.

3.3 Matrices

The matrices used in these experiments were downloaded from The University of Florida Sparse Matrix Collection [11], and chosen in order to ensure a broad selection of matrices, and hence, to ensure insight into what cases optimize the benefits of CSX. However, as was discussed in Section 2.2, CG has some limitations with regards to the problems that can be solved:

1. The matrix must be symmetrical.
2. The matrix must be positive definite.
3. The system must be solvable (i.e. the matrix must come with a b-vector).

Based on these constraints, 21 suitable matrices were found, which are presented in Table 3.7. The Table also contains the following parameters for each matrix:

rank is the number of linearly independent equations in the linear algebra system. As a matrix represents the completely specified linear system, this number is equal to the number of rows and columns in the matrix.

nonzeroes is the number of non-zero values in each matrix.

sparsity is the number of *nnz*s per element in the matrix ($\frac{nnzs}{rank^2}$).

footprint is the size of the *nnz*s assuming double precision floats (64-bit). Double precision floats are used in the CG application, and hence, this number indicates how much memory is needed to store the raw data of the matrix.

condition is the condition number for the given matrix. This number indicates how well each iteration of CG converge on the solution vector, and its definition can be found in Section 2.2.4. The number was estimated for each matrix with the MATLAB-function *condest()*.

In addition to Table 3.7, Appendix A contains the structure of the matrices. These plots are provided alongside the matrices from the Matrix Collection [11], and show how the *nnz*s are distributed. This unveils structural properties for the matrices, which can be used to enhance the studies of their performance and energy efficiency.

In general, we see that most of these matrices are relatively small (> 20MB). This means that they will fit in the LLC (see Table 3.1) of the target machines, reducing costly memory accesses. On one hand, this will most likely increase the

Name	rank	nonzeroes	sparsity	footprint	condition
2cubes_sphere	101492	874378	8.49×10^{-5}	6.67MB	2.9388e+09
af_5_k101	503625	9027150	3.56×10^{-5}	68.87MB	6.4275e+08
af_shell3	504855	9046865	3.55×10^{-5}	69.02MB	1.4403e+06
bone010	986703	36326514	3.73×10^{-5}	277.15MB	1.2165e+09
boneS01	127224	3421188	2.11×10^{-4}	26.10MB	4.2170e+07
boneS10	914898	28191660	3.37×10^{-5}	215.09MB	2.9528e+08
gyro	17361	519260	1.72×10^{-3}	3.96MB	3.4851e+09
LF10000	19998	59990	1.50×10^{-4}	0.46MB	6.4743e+18
nasa2146	2146	37198	8.08×10^{-3}	0.28MB	4.1303e+03
nasa2910	2910	88603	1.05×10^{-2}	0.68MB	1.7650e+07
nasa4704	4704	54730	2.47×10^{-3}	0.42MB	1.6576e+08
nasasrb	54870	1366097	4.54×10^{-4}	10.42MB	1.4836e+09
olafu	16146	515651	1.98×10^{-3}	3.93MB	2.2532e+12
offshore	259789	2251231	3.34×10^{-5}	17.18MB	2.3284e+13
parabolic_fem	525825	2100225	7.60×10^{-6}	16.02MB	2.1108e+05
Pres_Poisson	14822	365313	1.66×10^{-3}	2.79MB	3.1983e+06
raefsky4	19779	674195	1.72×10^{-3}	5.14MB	1.5172e+14
smt	25710	1889447	2.86×10^{-3}	14.42MB	6.1260e+09
sts4098	4098	38227	2.28×10^{-3}	0.29MB	4.5095e+08
thermal1	82654	328556	4.81×10^{-5}	2.51MB	4.9625e+05
thermal2	1228045	4904179	3.25×10^{-6}	37.42MB	7.4806e+06

Table 3.7: Matrices used in experiments

performance of the CG execution. On the other, the matrices will most likely not benefit significantly by the memory optimizations of CSX. There are also some fairly large matrices (e.g. bone010), that will not fit in the LLC and thus have quite different performance compared to the smaller ones.

The matrix set constitutes a wide set of ranks ([2146, 1228045]), *nnz* counts ([37198, 28191660]), sparsities ([1.05×10^{-2}, 7.60×10^{-6}]) and condition numbers ([4.1303e+03, 6.4743e+18]). This ensures that the set covers most matrix properties that can affect performance.

Chapter 4

Results

In this Chapter, the results of the runs described in Chapter 3 are presented. The Chapter is divided into Sections for each of the properties that are to be examined.

As was presented in Section 3.2.2, the applications executed in these experiments can be run in several different configurations. In order to easily distinguish between these CPU and thread configurations, a short description format has been chosen. This format is inspired by the similar classification used by Karaksis *et al.* in their evaluation of energy trade-offs for CSX [23].

This format is as follows: $C\text{c}T\text{t}S\text{s}X$, where C denotes how many physical CPUs are used, T denotes how many threads are used, S denotes how many threads are run as HyperThreads (SMT) and X indicates what algorithm has been used (R for CSR, X for CSX and Xs for CSX with statistical sampling (see Section 2.4.1)).

As an example, the description *2c8t0sX* indicates that the application has been run on two physical CPUs (i.e. two sockets), it has been run with eight threads divided between these two CPUs (four threads per CPU) and none of these threads have been run as HyperThreads using SMT. The application has been run using CSX.

Another example would be *2c32t16sXs*, which indicates that two physical CPUs was used, 32 threads were divided among these two CPUs, 16 of these threads (eight per CPU) were run as HyperThreads using SMT and the application was run using CSX with statistical sampling enabled.

This Chapter is structured into four Sections: Section 4.1 contains a description of the different formats used and their effect on execution and energy consumption. Section 4.2 contains a description of the matrix set and results on how the different matrices affect the energy consumption of CSX. Similarly, Section 4.3 contains the results of how different hardware configurations and properties affect CSX, while Section 4.4 contains results on how the Vilje supercomputer compares to the CARD-server for execution of CG.

4.1 Format properties

This Section presents the differences between the formats used in this work to perform SpMV. The formats in question are CSR, the simplest of them, CSX, the compression optimization of CSR, and CSX with statistical sampling, a preprocessing optimization for CSX.

First, the characteristics of each different format will be presented. For each of them, the power-time plot will be examined and analyzed in order to gain insight into how the execution behaves with regards to energy consumption. This is closely related to preliminary studies of CSX's energy efficiency [21], as well as the work of Meyer *et al.* [28].

Thereafter follows a comparison of the energy efficiency of the different formats for both the CG execution and the full application execution.

4.1.1 Energy profile

In this part, the power-time plots of each of the configurations are shown, different phases of execution are analyzed and the overall power consumption discussed. The Section is split into three parts: CSR, CSX and CSX with statistical sampling.

For the plots shown in this Section, a four threaded run of the matrix *af_5_k101* will be displayed. This is done for four reasons:

1. The runtime of *af_5_k101* is long enough for the phases of execution to clearly be shown in the plots.
2. The power-time plots of *af_5_k101* show an average profile with few anomalies compared to the other matrices, making them well suited for general discussion.
3. A run with four threads shows the effect of parallelism, and will for this reason be better suited to show the different phases of execution than runs with smaller numbers of threads.
4. *af_5_k101* did function properly with the use of statistical sampling. Some of the matrices did not, and hence are not usable for such a comparison. This is elaborated further in the Subsection about CSX with statistical sampling.

The plots contain both results from the RAPL interface in the MSR, as well as results from the Yokogawa wall power meter. This makes us able to investigate how well the CPU power consumption and wall power consumption correlate, and thus, how well we can extrapolate results from the MSR when discussing the overall energy consumption of the machine.

CSR

CSR is the simplest of the formats, merely describing a way of storing only the *nnz* values of a sparse matrix. As described in Section 2.3, it forms the basis for CSX.

In Figure 4.1, the power-time plot of a four threaded execution of CSR on the matrix *af_5_k101* is presented.

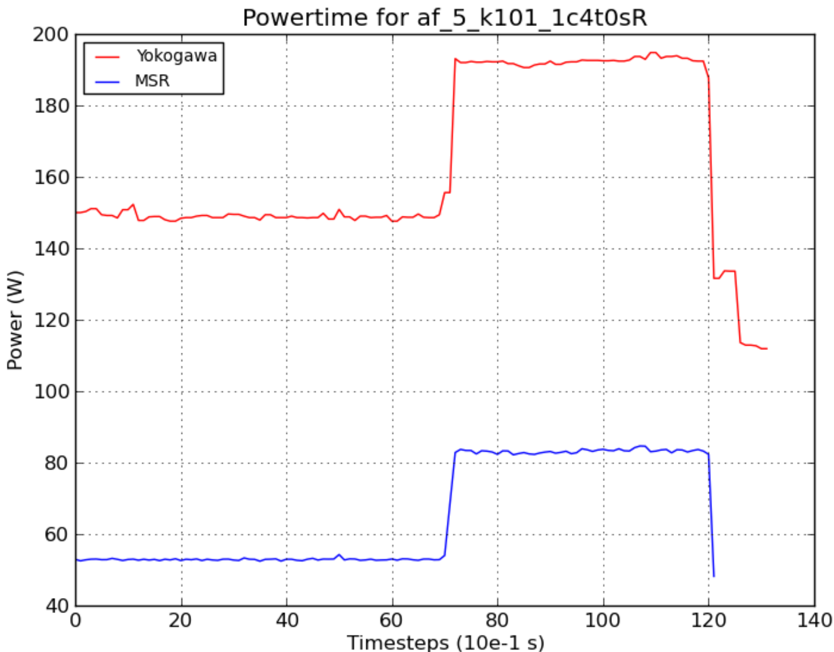


Figure 4.1: Power-time plot for four threaded af_5_k101 CSR

The execution of CG with CSR is, as we can see from the plot, split into two phases. The first phase, lasting for about seven seconds at 55W/150W, was in earlier work identified as the *set-up and fetch phase, where the matrix is retrieved from disk and stored to memory in the CSR scheme* [21].

The second phase of execution, lasting for approximately 5 seconds at 85W/190W was identified as the *actual execution phase* [21]. This is where the 1024 CG iterations are executed.

Apparent from this plot is the fact that only the second phase is affected by the parallelization offered by the four threads. This is to be expected, as the first phase consists of serial access to memory, which cannot be parallelized. The second phase utilizes the DLP found in SpMV to gain performance through parallelization. This can be seen as the increase in power for this phase. As this has been thoroughly

analyzed, the reader is referred to earlier research [21] for more discussion on the subject.

Looking at the comparison between the MSR and Yokogawa, we notice a distinct correlation between the graphs. Once the CPU starts consuming additional power during the CG execution, the power used by the entire system rises correspondingly. However, a shift can be observed in the difference between the measurements. In the first phase of execution, the power difference is approximately 95W, while in the second phase, this difference increases to about 105W. This implies that not only the additional power consumption of the CPU affects the full system power, but causes other parts of the system to consume additional power as well. A likely assumption is that the increased activity of the CPU causes an increase in the power consumption of the memory system. However, this can not be confirmed without measuring the power consumption of the memory system, which was not available for this work.

CSX

CSX, as stated in Section 2.4, is built on the same principles as CSR, but utilizes preprocessing and compression in order to increase performance. While the performance gain can be significant, it comes at the cost of the preprocessing.

In Figure 4.2, the power-time plot of the execution of CSX on the matrix *af_5_k101* is presented.

This plot is, when compared to the plot of CSR, significantly more convoluted. In spite of this, several unique phases can be identified. In earlier work, these were characterized as three phases: the *set-up and fetch phase*, the *delta encoding phase* and the *execution phase* [21]. However, the more recent work for Meyer *et al.* correctly characterized five distinct phases [28]:

1. Loading of the matrix from the disk (single-threaded)
2. Substructure detection (multithreaded)
3. Matrix encoding (multithreaded)
4. Code generation (single-threaded)
5. Sparse matrix-vector kernel (multithreaded)

Not all of these phases are easily distinguished in the plot. However, some results of the different characteristics of these phases can be found. The first phase, which is equal to the first phase of CSR, can be seen in the first seven seconds of the plot. The second phase then follows with major fluctuations in the power. This phase lasts from approximately the seven second mark to the 30 second mark, which is majority of the plot. Phase three then ensues, with notably smaller fluctuations and a notable decrease in power compared to phase two. This phase approximately lasts from the 30 second mark to the 34 second mark. The fourth phase can be

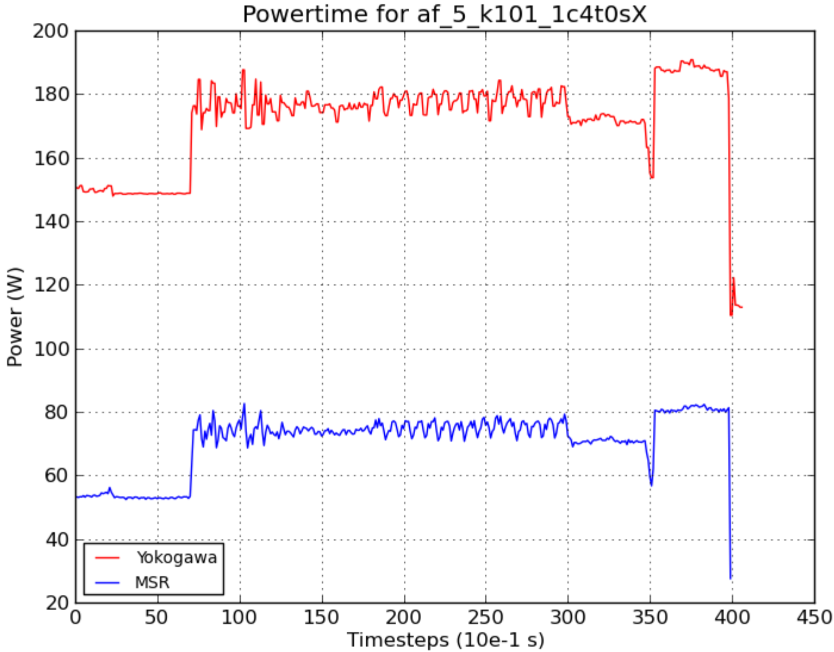


Figure 4.2: Power-time plot for four threaded af_5_k101 CSX

seen in the narrow trough close to the 35 second mark. The final phase, which is equal to the second phase of CSR, can be seen from the 35 second mark to the 40 second mark.

One major thing to notice is that the preprocessing consumes most of the runtime and thus energy for this execution. As this has been discussed in earlier work, the reader is referred to Simonsen [21] for a review of this topic and its implications.

Regarding the comparison between MSR and Yokogawa, this plot, as the plot in Figure 4.1, shows a high degree of correlation. This is especially apparent in phase two, where the fluctuations can be equally observed for both graphs. Minor discrepancies can be found, but they are not large or frequent enough to be substantial. The pattern of increase in CPU power consumption causing additional increase in full system power can also be observed in this plot.

CSX with statistical sampling

In previous work [21], one of the shortcomings was the inability of using the statistical sampling for preprocessing offered by CSX. This issue, however, has since partly been resolved by a patch for CSX. Due to statistical sampling now being

available, this Section contains a discussion of differences when using statistical preprocessing compared to runs of regular CSX.

In Figure 4.3 the power-time plot equivalent to the previous plots for CSR and CSX is shown.

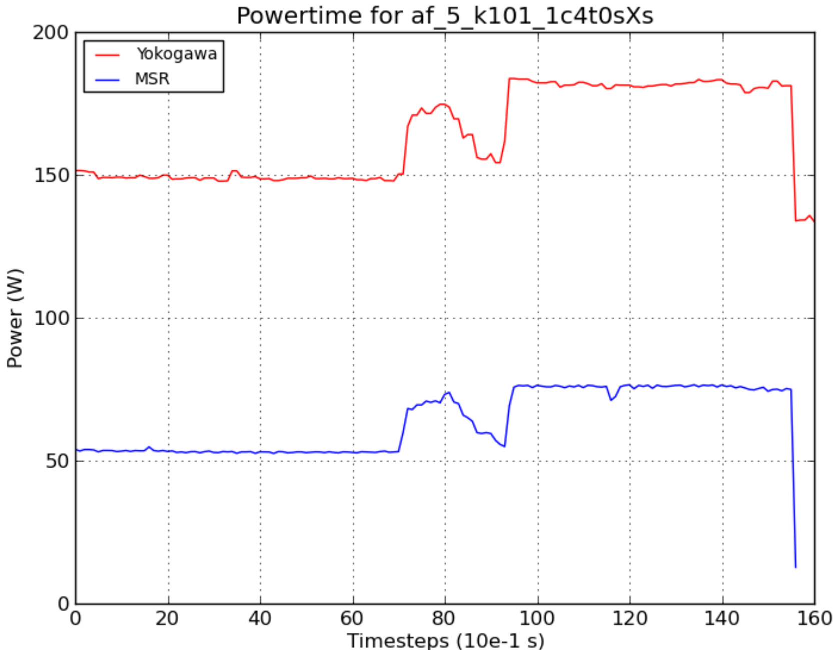


Figure 4.3: Power-time plot for four threaded af_5_k101 CSX w/SP

Because the statistical sampling is but an optimization for the CSX preprocessing, the same phases of execution can be found in this plot. However, when comparing this to the plot of CSX, one important observation stands out: the time consumed by the preprocessing is drastically reduced. While the preprocessing of CSX for this matrix lasted for about 28 seconds, the equivalent preprocessing of CSX with statistical sampling lasts for only two seconds. This improvement is due to only one percent of the *nnz*s being searched when using statistical sampling (see Section 3.2.2). However, this improvement might cause a decrease in performance for the CG execution, as discussed in Section 2.4.1. This will be further investigated in Section 4.1.2.

The three phases of the preprocessing mentioned in the discussion of CSX above become somewhat more indistinguishable in this plot. The second phase can be found between seven and eight seconds, the third phase can be found between eight and nine seconds, while the fourth phase can be found at approximately nine seconds. When compared to the same phases of regular CSX, we see that the

runtime phase two and three are heavily decreased. For the fourth phase, however, the same can not be confirmed nor rejected from these plots.

Like in the previous plots, the MSR and Yokogawa graphs show high degrees of correlation with the same properties as pointed out earlier.

Even though the statistical sampling is showing promising results for the overall performance of CSX, there is one major issue that must be discussed. When using CSX with statistical sampling on the matrix set presented in Section 3.3, it was found to cause an assert error in the application, causing it to terminate. This issue appeared for all matrices shown in Table 4.1.

Matrix
LF10000
nasa2146
nasa2910
nasa4704
olafu
Pres_Poisson
raefsky4
sts4098

Table 4.1: Matrices that caused CSX with statistical sampling to terminate

Due to this issue severely limiting the matrix set, the statistical sampling will not be taken into consideration for the rest of the results in this Chapter, with exception of Section 4.1.2. This is done in order to ensure a broad set of matrices that cover the most possible corner cases in the results.

4.1.2 Comparison

In this Section, the formats examined above will be compared. The comparison will be done for both the CG execution phase and the full execution of the application, looking at both runtime and energy consumption. As was pointed out in the previous Sections, the correlation between the MSR and Yokogawa was found to be high. Because of this, and the fact that some of the executions of the smaller matrices were too short for the Yokogawa results to be accurate, MSR measurements will be used for comparison between the formats.

As shown in Table 4.1, not all matrices are eligible for statistical sampling. Due to this, the plots in this Section will contain the average runtime and energy consumption of all matrices for CSR and regular CSX, as well as the average runtime and energy consumption of the eligible matrices in all three formats.

CG execution

First, the CG execution will be examined. Figure 4.4 contains the average runtime and energy consumption plots for the CG execution.

These plots show, as one would expect, that increased parallelization causes increased performance. However, four significant observations can be made:

1. There is a close correlation between runtime and energy consumption. The only notable difference is for CSR when moving from four to eight threads, where the energy consumption increases while the runtime decreases. Note also that the ordering of the formats is closely matched for all configurations.

2. CSX starts outperforming CSR in average energy consumption for four threads and runtime for eight threads. This has two implications: one should use a high number of threads when running CSX (further elaborated in Section 4.3.1), and the memory optimization of CSX becomes profitable between four and eight threads, showing that this is where the memory bottleneck starts limiting the application.

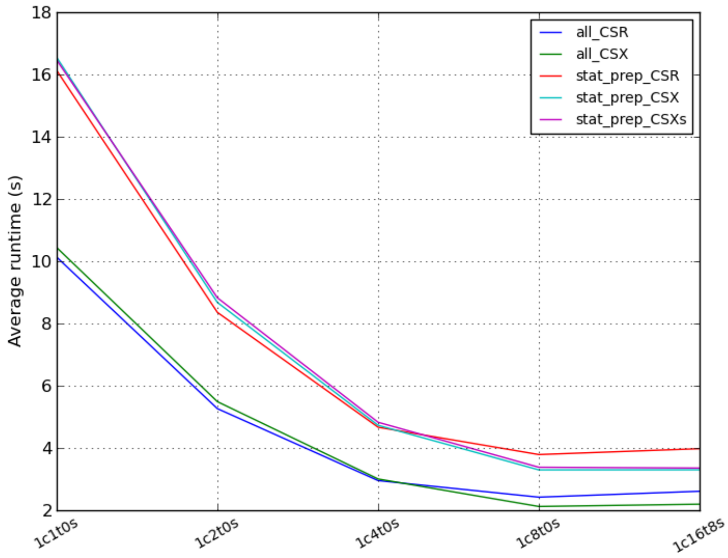
3. CSR has its lowest energy consumption at four threads. When adding four additional threads, the energy consumption increases, even though the runtime continues to decrease. This is not the case for CSX with and without statistical sampling, which shows a reduction in both runtime and energy consumption. This behavior will be further examined in Section 4.3.1.

4. When increasing the thread number from eight to 16, there is a slight increase in runtime and energy consumption for all configurations. However, the 16 threaded executions are run on a single core with eight HyperThreads. This can have a negative effect either because of the use of SMT, which in earlier work has been shown to negatively affect both runtime and energy consumption [21], or because the 16 threads running on a single core can cause congestion and thus decreased performance. The latter will also be further examined in Section 4.3.1.

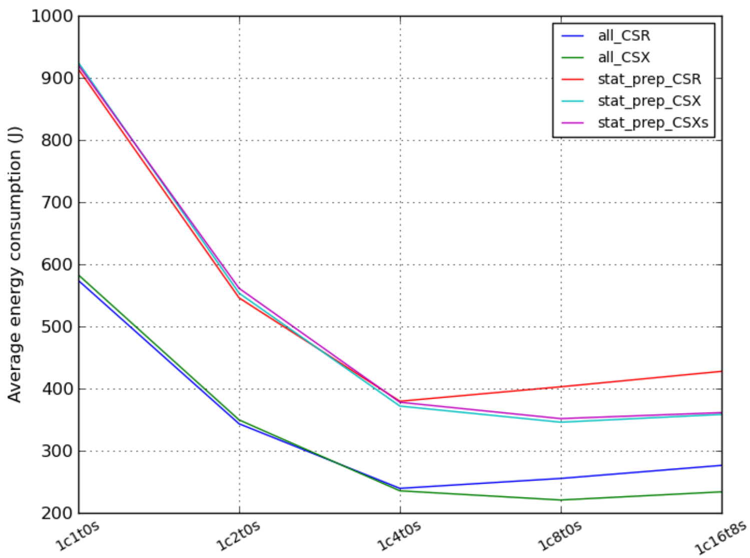
Based on these results, we can conclude that CSX should be used with respect to both performance and energy efficiency given four threads or more. We also see that statistical sampling results in slightly higher runtime and energy consumption, as expected. Note that these results are only for the actual CG execution phase, and thus do not incorporate the preprocessing of CSX.

Full application energy

In this Section, the runtime and energy consumption of the full application execution will be examined. The plots in Figure 4.5 show the appropriate average graphs, similar to the plots in the previous Section. As was noted in the introduction to this Section, these results are based on MSR measurements.

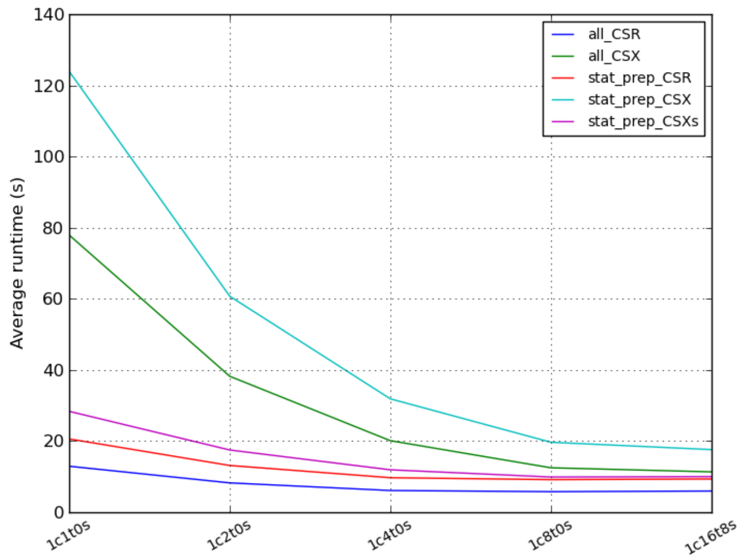


(a) Runtime

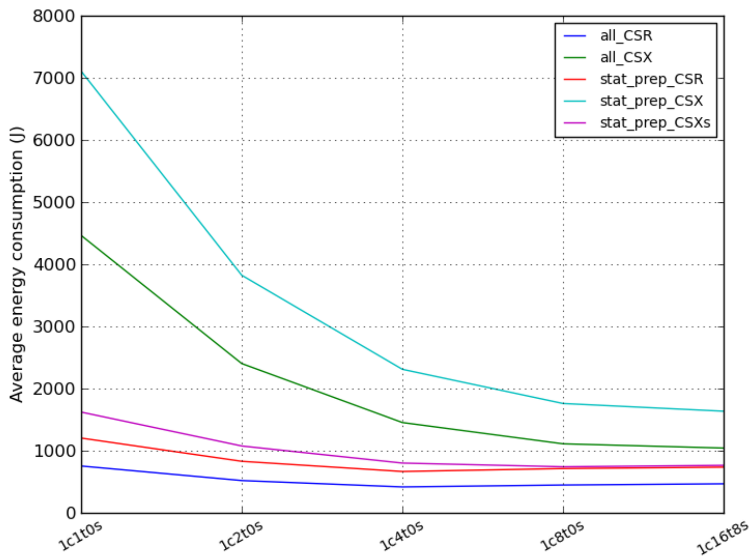


(b) Energy

Figure 4.4: Comparison of CG execution



(a) Runtime



(b) Energy

Figure 4.5: Comparison of full application execution

From these plots we clearly see the cost of CSX's preprocessing. For all thread configurations, CSR shows significantly lower runtime and energy consumption than CSX. This complies with previous research on the topic [21]. New to this work is the inclusion of statistical sampling in the comparison.

As was shown in Section 4.1.1, the statistical sampling optimization drastically reduces the runtime of the preprocessing. This is shown in the plots above, where CSX with statistical sampling is close to the performance and energy consumption of CSR, while CSX with full preprocessing shows significantly higher results.

From these plots, it also becomes clear that CSX, regardless of preprocessing technique, shows better parallelization properties than CSR. While the improvement for CSR is marginal from one to four threads and insignificant for further increase in threads, CSX in both preprocessing configurations shows great improvement in both performance and energy efficiency, although the improvement is decreasing as more threads are added.

For eight and 16 threads, CSX with statistical sampling shows only marginally higher runtime and energy efficiency than CSR for the full execution of the application, showing that CSX in this configuration almost overcomes its preprocessing with the gain in CG execution.

All of the executions shown in this Section are based on 1024 CG iterations, which, as discussed in Section 3.2.2, is fewer than would realistically be used to solve CG. This, combined with the results for CG execution shown in the previous Section, show that CSX might potentially save energy when compared to CSR, and that CSX with statistical sampling will save energy.

4.2 Matrix properties

In this Section, the effect of matrix properties on performance and energy consumption are examined. Based on the results, the matrices are categorized into groups and analyzed to gain knowledge about what matrices CSX is suited for.

The Section will first look at energy consumption properties of the matrices through discussion of energy consumption per *nnz* value. Then follows a discussion of energy optimizability through CSX, based on a comparison between CSR and CSX.

4.2.1 Energy per nonzero

In order to determine how the properties of the matrices affect the energy consumption of CG execution, the results of the energy measurements are normalized with regards to the size of the matrices, producing a number for energy consumed per *nnz* value. This number in then used to compare the performance of the matrices in order to categorize them.

The results are presented in Figure 4.6. Figure 4.6(a) contain the results for CSR, and Figure 4.6(b) for CSX.

From these plots, we can already identify three distinct groups of matrices. The first group, consisting of the matrices shown in Table 4.2, share a pattern in which they have a fairly flat progression until they reach eight threads, at which point the energy/*nnz* begins to increase with the number of threads. These matrices are also among the matrices that consume the most energy/*nnz*, regardless of configuration.

Name	**rank**	**nonzeroes**	**sparsity**	**footprint**	**condition**
LF10000	19998	59990	1.50×10^{-4}	0.46MB	6.4743e+18
nasa2146	2146	37198	8.08×10^{-3}	0.28MB	4.1303e+03
nasa2910	2910	88603	1.05×10^{-2}	0.68MB	1.7650e+07
nasa4704	4704	54730	2.47×10^{-3}	0.42MB	1.6576e+08
sts4098	4098	38227	2.28×10^{-3}	0.29MB	4.5095e+08

Table 4.2: low_nnz_group

When looking at the matrix data for the matrices in this group, presented in Section 3.3, we see that these matrices share one common denominator: they have the smallest number of *nnz*s, with less than 100.000 values. Another notable observation is that the order in which they are found for eight and 16 threaded configuration. When looking at energy/*nnz* from highest to lowest, their ordering is exactly equal to their order when looking at the number of *nnz*s from lowest to highest.

This suggests that matrices with low numbers of *nnz*s perform worse than other matrices for both CSR and CSX, regardless of their structure. Because of this, the group consisting of the matrices listed above will henceforth be known as the *low_nnz* group.

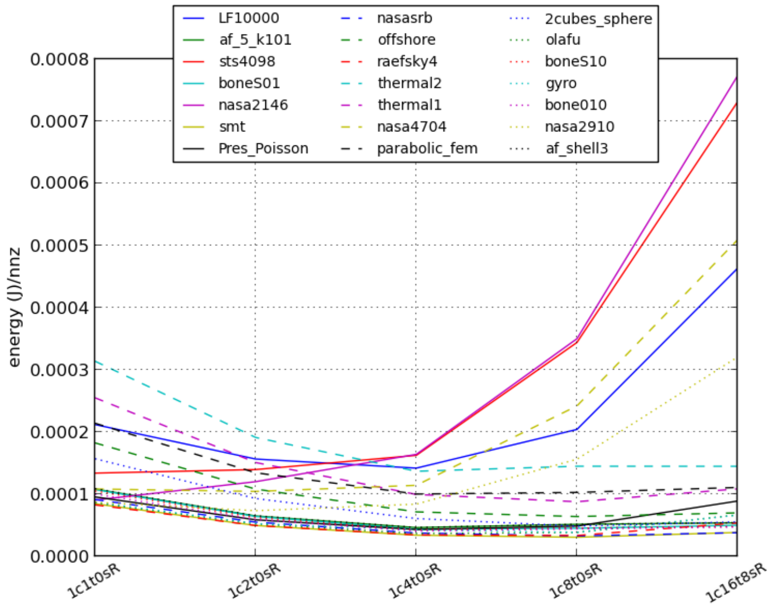
Disregarding the *low_nnz* group, we see that the rest of the matrices fall into either of two groups: those that fall within the bottom belt of matrices and those that perform worse.

The latter of these two groups, henceforth known as the *middle_group*, consists of the matrices shown in Table 4.3. These matrices, as noted above, showed sufficiently poor performance when compared to the other matrices to be categorized as a distinct group.

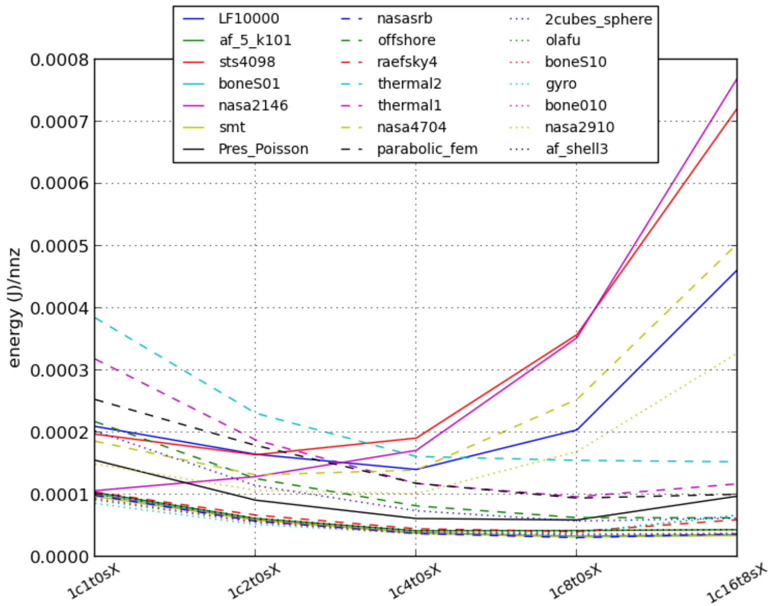
In Figure 4.7, the energy/*nnz* plot of the matrices that form the *middle_group* is shown.

The general, we see that the energy/*nnz* for matrices in this group favors CSR. It is first at eight threads CSX outperforms CSR for this metric, and then only for the matrices *offshore* and *parabolic_fem*. This implies that the memory optimization of CSX does not prove beneficial for these matrices in general.

When looking at the data of Table 4.3, we see that none of these matrices show any



(a) CSR

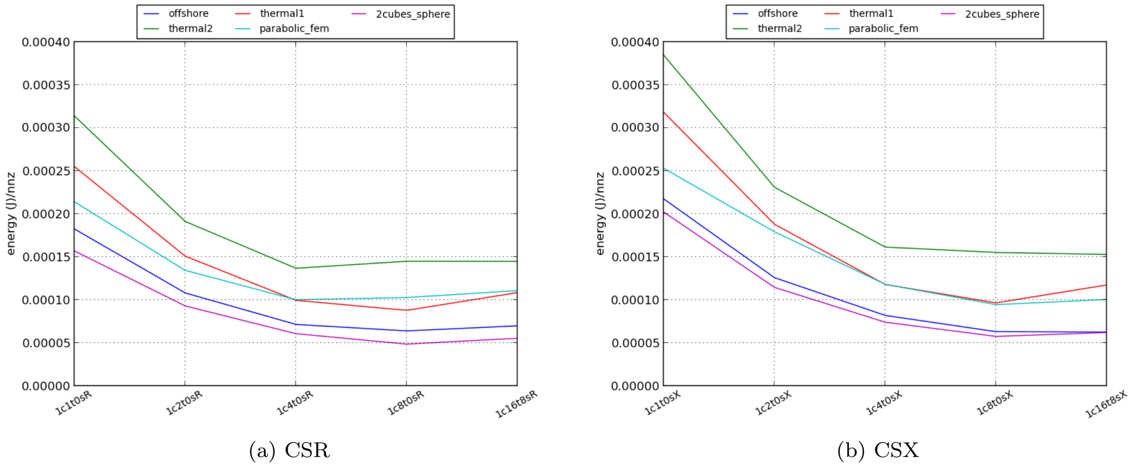


(b) CSX

Figure 4.6: Energy/nnz for all matrices

Name	rank	nonzeroes	sparsity	footprint	condition
2cubes_sphere	101492	874378	8.49×10^{-5}	6.67MB	2.9388e+09
offshore	259789	2251231	3.34×10^{-5}	17.18MB	2.3284e+13
parabolic_fem	525825	2100225	7.60×10^{-6}	16.02MB	2.1108e+05
thermal1	82654	328556	4.81×10^{-5}	2.51MB	4.9625e+05
thermal2	1228045	4904179	3.25×10^{-6}	37.42MB	7.4806e+06

Table 4.3: middle_group



(a) CSR

(b) CSX

Figure 4.7: Energy/nnz for middle matrices

distinct similarities in properties that stand out when compared to other matrices. They vary in rank, *nnz*s and condition number, but are found within the same general area of sparsity. This property, however, does not explain their performance as many of the other matrices also fall within the same area.

As no explanation for these results can be found in the matrix properties of Table 3.7, they must be caused by other properties measured with other metrics. From Figure A.1, we see that the matrices share one property: they all contain quite a few corner values (i.e. values near the top right and bottom left corners of the matrix). Besides this, we see that they also contain values spread throughout the matrix. Some other matrices (namely *Pres_Poisson*, *raefsky4*, *smt* and *sts4098*) share this property, but they have considerably lower rank and sparsity than the matrices of the *middle_group*.

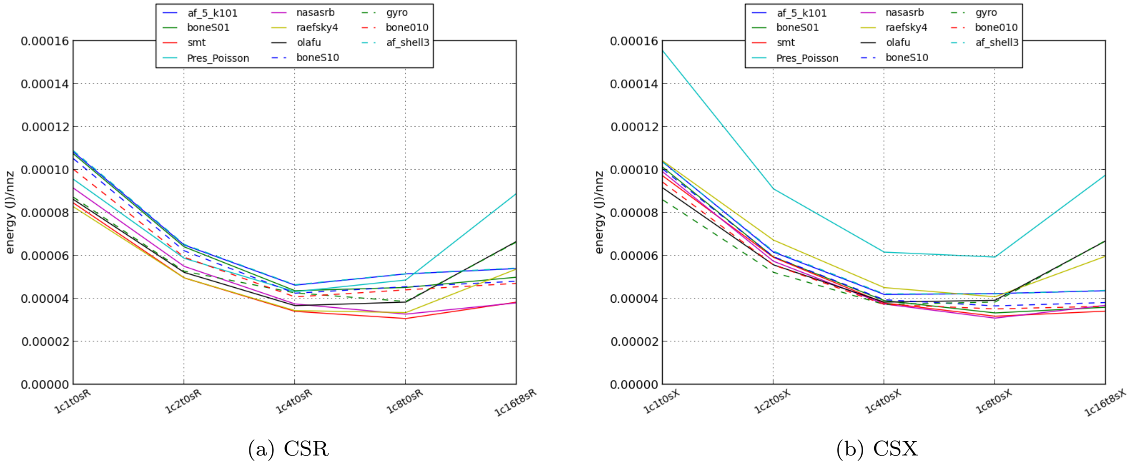
Based on this, we can argue that sparse values near the non-diagonal corners of the matrices cause the energy efficiency of the CG execution to drop. This also causes the effect of the CSX optimization to diminish, implying that CSR should be used for these kinds of matrices.

The remaining matrices fall within the group henceforth known as the *belt_group*. These matrices perform similarly and well compared to the other groups, hence the name. The matrices making up this group are shown in Table 4.4

Name	**rank**	**nonzeroes**	**sparsity**	**footprint**	**condition**
af_5_k101	503625	9027150	3.56×10^{-5}	68.87MB	6.4275e+08
af_shell3	504855	9046865	3.55×10^{-5}	69.02MB	1.4403e+06
bone010	986703	36326514	3.73×10^{-5}	277.15MB	1.2165e+09
boneS01	127224	3421188	2.11×10^{-4}	26.10MB	4.2170e+07
boneS10	914898	28191660	3.37×10^{-5}	215.09MB	2.9528e+08
gyro	17361	519260	1.72×10^{-3}	3.96MB	3.4851e+09
nasasrb	54870	1366097	4.54×10^{-4}	10.42MB	1.4836e+09
olafu	16146	515651	1.98×10^{-3}	3.93MB	2.2532e+12
Pres_Poisson	14822	365313	1.66×10^{-3}	2.79MB	3.1983e+06
raefsky4	19779	674195	1.72×10^{-3}	5.14MB	1.5172e+14
smt	25710	1889447	2.86×10^{-3}	14.42MB	6.1260e+09

Table 4.4: belt_group

Figure 4.8 shows the energy/nnz plot for the matrices in this group. For these matrices, we see that there are some variations between CSR and CSX, most notably for the matrices *Pres_Poisson* and *raefsky4*. These two matrices perform notably worse for CSX than CSR. For the other matrices, however, the variations are too small to be conclusive.



(a) CSR (b) CSX

Figure 4.8: Energy/nnz for belt matrices

When looking for a common denominator for this group of matrices, Table 3.7 again shows that the matrices represent a broad range of properties. Hence, no conclusions can be drawn.

When looking at Figure A.1, however, we see that most of the matrices are simple in their structure, mostly consisting of values along the diagonal. There are some exceptions to this, namely *smt*, *Pres_Poisson*, *raefsky4* and to some degree *gyro*. This is interesting, as these four matrices share a low rank and sparsity. As was discussed for three of these matrices during the categorization of the *middle_group*, their low rank and sparsity cause them to generally perform better than their structure implies that they should. Hence, these four matrices would probably be found in the *middle_group*, had their rank and sparsity been greater.

Based on these results and categorizations, we can argue that matrices that are sufficiently large and do not have a lot of spread, non-diagonal corner values, are well suited for energy efficient solving of CG in general.

4.2.2 Energy consumption optimizability through CSX

While energy per *nnz* gives insight into how well matrices perform for CG, it does not effectively show how well the matrices are optimized by CSX. To achieve this, this Section presents plots of the ratio between energy consumption in CSX and CSR. This ratio, henceforth known as the Energy Decrease Ratio (EDR), is created by dividing the CSR energy consumption by the CSX energy consumption ($\frac{E_{csr}}{E_{csx}}$). This makes it possible to identify key properties in the matrices that affect the energy efficiency of CSX.

Figure 4.9 contains the EDR plots for all matrices. The grid lines for this plot have been removed in order to improve readability for the dotted lines.

The most apparent feature of this plot is that the EDR increases with the number of threads (i.e. towards the right), again showing that CSX parallelizes better than CSR.

When examining the matrices, we notice that a small group of matrices show energy decrease (EDR of 1.0 or above) for all configurations, another group show decrease for some configurations, while a large group show no improvement from using CSX over CSR.

The first of these groups, consisting of the matrices shown in Table 4.5, benefits greatly from the optimizations of CSX. This group will therefore henceforth be called the *csx_optimized_group*. When looking for common denominators, two observations become apparent. Table 3.7 shows that they are all among the top matrices in *nnz* count, with footprints larger than the LLC of 20MB. Because of this, none of them can be kept on chip during runtime, causing additional memory accesses during execution.

These results can be readily explained by CSX providing a memory optimization, and thus provide more optimization for larger matrices that have greater need of the memory bus. An important note here, however, is that other large matrices, such as *thermal2* and *parabolic_fem*, are not among these matrices. Hence, the number of *nnz*s can not solely describe the effectiveness of CSX.

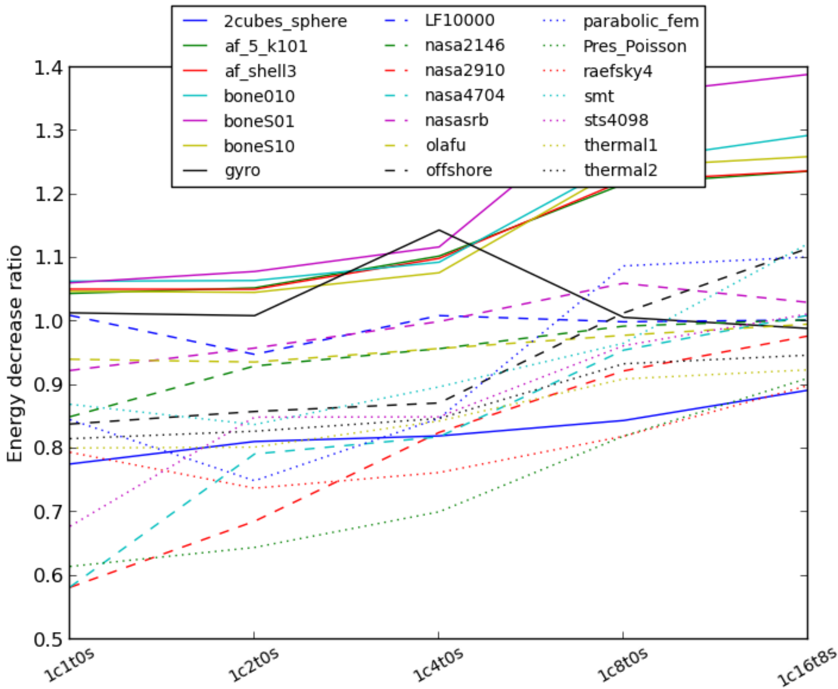


Figure 4.9: EDR for all matrices

Name	rank	nonzeroes	sparsity	footprint	condition
af_5_k101	503625	9027150	3.56×10^{-5}	68.87MB	6.4275e+08
af_shell3	504855	9046865	3.55×10^{-5}	69.02MB	1.4403e+06
bone010	986703	36326514	3.73×10^{-5}	277.15MB	1.2165e+09
boneS01	127224	3421188	2.11×10^{-4}	26.10MB	4.2170e+07
boneS10	914898	28191660	3.37×10^{-5}	215.09MB	2.9528e+08

Table 4.5: csx_optimized_group

The second denominator for this group is the structure of the matrices. From Figure A.1, we see that all matrices in this group have values centered around their diagonal. This makes the patterns of values easily identifiable by CSX, as discussed in Section 2.4, and thus easily optimizable. Only one other matrix apart from this group shares this property, namely *LF10000*, which performs notably worse than the group. This, however, can be explained by the small number of *nnzs* in this matrix, with the same discussion as in the previous paragraph.

These observations suggest that in order to decrease energy consumption by using CSX, one will have to ensure that the input matrices are sufficiently large, preferably larger than the LLC size, with a diagonal matrix structure.

The second group of matrices, hereafter called the *semi_optimized_group* consists of matrices that benefit from CSX with regards to energy consumption in some configurations, but not all. The matrices that make up this group is shown in Table 4.6.

Name	**rank**	**nonzeroes**	**sparsity**	**footprint**	**condition**
gyro	17361	519260	1.72×10^{-3}	3.96MB	3.4851e+09
LF10000	19998	59990	1.50×10^{-4}	0.46MB	6.4743e+18
nasasrb	54870	1366097	4.54×10^{-4}	10.42MB	1.4836e+09
offshore	259789	2251231	3.34×10^{-5}	17.18MB	2.3284e+13
parabolic_fem	525825	2100225	7.60×10^{-6}	16.02MB	2.1108e+05
smt	25710	1889447	2.86×10^{-3}	14.42MB	6.1260e+09
sts4098	4098	38227	2.28×10^{-3}	0.29MB	4.5095e+08

Table 4.6: semi_optimized_group

In general, these matrices perform better (in EDR) for CSX as the number of threads increases. However, this does not apply to all the matrices. Looking at *gyro* and *LF10000*, we see that they are maximized for four threads, while *nasasrb* is optimized by eight threads. These are also the only matrices that have this property.

When looking for denominators for this group, Table 4.6 shows us that they differ in rank and *nnz* count. However, they are all smaller than the LLC size of the machine. Structurally, there are some similarities that can be found. From Figure A.1, we see that most of these matrices have elaborate memory patterns with values spread throughout the entire matrix. On the other hand, *LF10000* and *nasasrb* are seemingly diagonal, looking more like the matrices of the *csx_optimized_group*.

When compared to this group, however, these two matrices are much smaller, both in rank and *nnz* count. Thus, the argumentation used about size versus the memory optimization of CSX mentioned above can explain why these two matrices are in this group instead of the *csx_optimized_group*. In short, they are too small to sufficiently benefit from CSX, but large enough for the CSX optimizations to outweigh their overhead in certain configurations.

Regarding the remaining matrices of this group, they contain several grouped or linear/diagonal values well suited for CSX optimization. On the other hand, these grouped values (units) are spread throughout the matrices instead of being packed together, making for more units. This leads more accesses in order to read the entire matrix, minimizing the effect of delta encoding the indexes as the number of accesses closes in on the number needed to access all elements. This fact, combined with the small size of the matrices making them fit in the LLC, explains why they generally do not benefit from CSX.

The last group consists of matrices that show increased energy consumption for all configurations when using CSX. Therefore, the group will henceforth be known

as the *csr_optimized_group*. An overview of the matrices in this can be found in Table 4.7.

Name	**rank**	**nonzeroes**	**sparsity**	**footprint**	**condition**
2cubes_sphere	101492	874378	8.49×10^{-5}	6.67MB	2.9388e+09
nasa2146	2146	37198	8.08×10^{-3}	0.28MB	4.1303e+03
nasa2910	2910	88603	1.05×10^{-2}	0.68MB	1.7650e+07
nasa4704	4704	54730	2.47×10^{-3}	0.42MB	1.6576e+08
olafu	16146	515651	1.98×10^{-3}	3.93MB	2.2532e+12
Pres_Poisson	14822	365313	1.66×10^{-3}	2.79MB	3.1983e+06
raefsky4	19779	674195	1.72×10^{-3}	5.14MB	1.5172e+14
thermal1	82654	328556	4.81×10^{-5}	2.51MB	4.9625e+05
thermal2	1228045	4904179	3.25×10^{-6}	37.42MB	7.4806e+06

Table 4.7: csr_optimized_group

When looking for denominators for this group, we note that nearly all of these matrices, like the ones in the *semi_optimized_group*, are smaller than the cache size of the machine. Hence, the overhead vs optimization trade-off of CSX discussed earlier does not prove beneficial for these matrices. However, as this also holds for the *semi_optimized_group*, other factors separate these groups. Note that this is not the case *thermal2*, which is among the largest matrices of the set, well surpassing the size of the LLC.

Structurally, this group consist of mainly two types of matrices: the tridiagonal-based matrices of the *nasa* group and *olafu*, and the more elaborate structural matrices. Immediately, we see that the tridiagonal-based matrices perform significantly better than the others for high numbers of threads. For low numbers, however, the results are more mixed. This indicates that the tridiagonal matrices are better suited for the use of CSX as the resources on the chip become limited (i.e. as the number of threads reach and surpass the number of cores on the chip). This can be attributed to the same reasons as discussed above, with spread units limiting the performance gain of delta encoding due to the additional accesses required to fetch all the units.

This, however, requires the assumption that CSX scales better than CSR as the resources on chip become limited. As it has already been shown in Section 4.1.2, this is a safe assumption to make.

A more interesting discussion is why the matrices are distributed between this group and the *semi_optimized_group* shown in Table 4.6. With the exception of the matrices *gyro*, *LF10000*, *sts4098* and *thermal2*, a distinct difference between the matrices in the two groups can be observed: the matrices in the *semi_optimized_group* are larger than the matrices of the *csr_optimized_group*. This complies with the discussion above about CSX being more beneficial for larger matrices.

However, the issue of the four noncompliant matrices still needs to be resolved.

When looking at the matrix structure of these four matrices, several clues can be found. *LF10000* has a structure that matches the most CSX optimizable matrices in the set. This makes it well suited for the delta encoding of CSX, increasing its performance in spite of its small size. Similar argumentation can be used for *gyro* and *sts4098*, which contain many straight or diagonal values clusters than can be identified and optimized by CSX. Hence, their performance for CSX is higher than what their size indicates. For *thermal2*, opposite observations can be found, with value clusters that are neither diagonal nor straight. This, along with the many small value clusters spread throughout the matrix, minimizes the effect of the CSX optimizations, and hence, pushes the performance in favor of CSR.

4.3 Platform properties

This Section examines the platform specific properties that affect the performance and energy consumption of CSX. This is done to identify what machine setups are well suited for CSX executions.

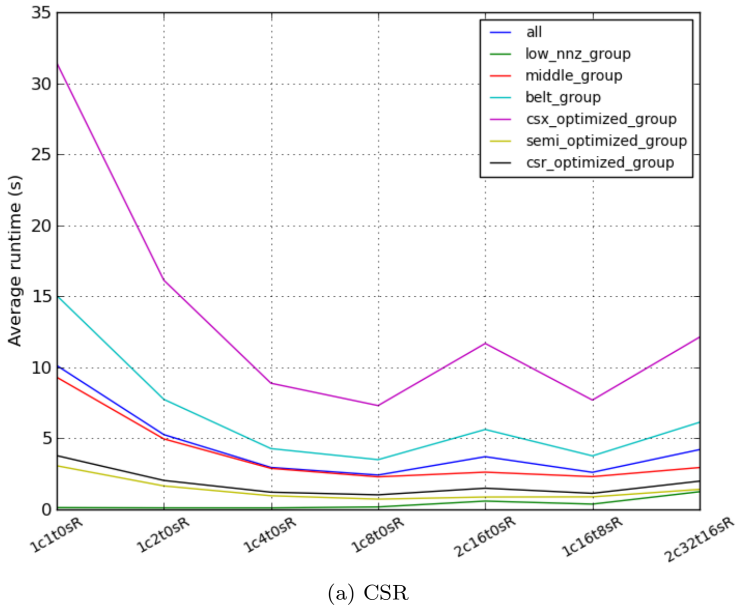
It contains the following parts: The first Section examines the effect of parallelization (i.e. how additional threads affect the energy consumption), the second Section examines the overhead and potential gains of using dual sockets, while the last Section explores the effect of clock frequency throttling.

4.3.1 Parallelization

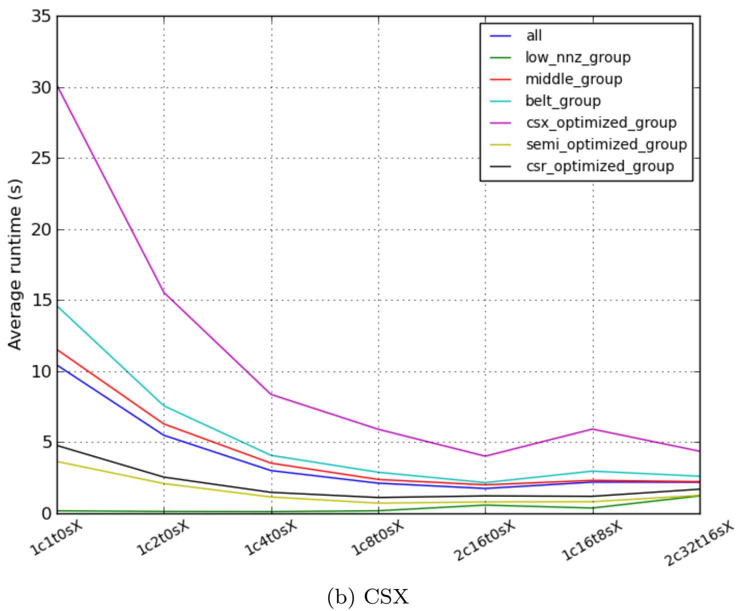
Parallelization through additional threads is a cheap and effective way of utilizing data level parallelism. As was noted in Section 2.1, SpMV is well suited for parallelization, and hence, we can expect both CSR and CSX to show performance improvements and energy consumption reductions with increased numbers of threads. This was also shown in previous work [21].

In this Section, the effect of parallelization on the general performance and energy efficiency will be examined. In addition, the effect of parallelization on the EDR, i.e. to what degree parallelization affects CSX compared to its effect on CSR, will be explored.

The Figures 4.10 and 4.11 contain the runtime and energy plots for the CG execution, respectively. The thread configurations shown are chosen in order to examine how additional threads affect the results. However, going beyond eight threads, either HyperThreading (SMT), two physical CPUs, or both will have to be used due to the eight cores on each CPU. As this potentially can affect the performance and energy efficiency (HyperThreads have previously been shown have a negative effect [21], while dual sockets will be explored in Section 4.3.2), both these configurations are shown in the plots for thread combinations that require either. This allows for general discussion of parallelization without having the results affected by one of these multithreading technologies.

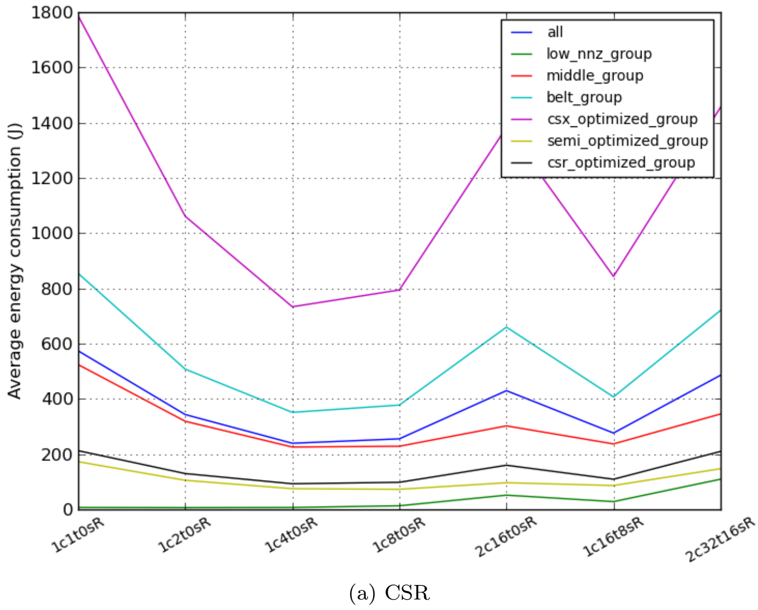


(a) CSR

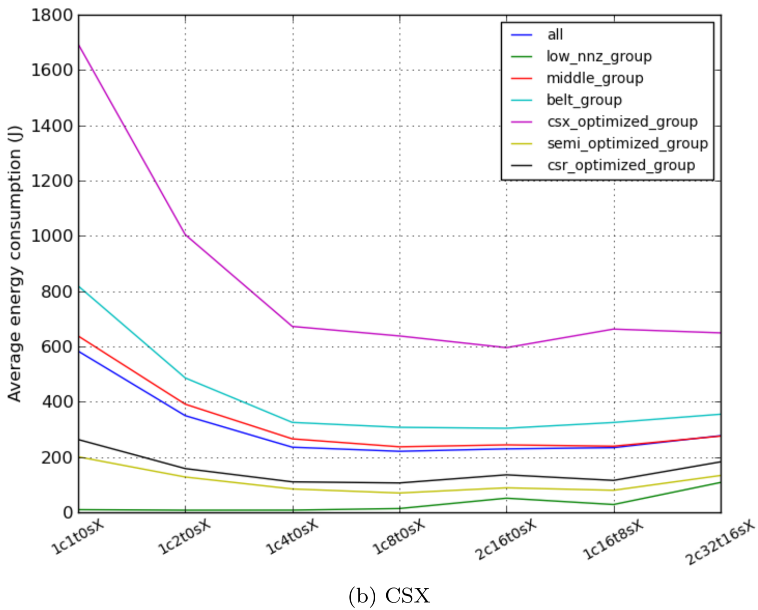


(b) CSX

Figure 4.10: Effect of parallelization on runtime



(a) CSR



(b) CSX

Figure 4.11: Effect of parallelization on energy

In general, we see that the use of multiple threads is beneficial for the configurations shown, including the averages of all matrices. This has already been established earlier and is due to the high degree of DLP, as discussed in Section 2.1. Beyond this generalization, however, some important observations can be made:

1. For energy efficiency, there is little to no benefit from increasing the number of threads beyond eight. Only the *csx_optimized_group* and marginally the *belt_group* shows continued decrease in energy consumption when moving beyond eight threads, but only for the 2c16t0sX configuration. All other matrix groups and configurations, including the average for all matrices, show an increase in energy consumption beyond eight threads. For CSR, four threaded configurations provide the optimal energy efficiency for all groups except the *semi_optimized_group*, which marginally benefits from eight threads.

   For runtime, similar results are apparent, but several groups are optimized for thread configurations beyond those that optimized their energy efficiency. For CSR, this is generally found for eight threaded configurations, while for CSX, it is generally found for 16 threaded configurations on two sockets.

   As was noted earlier in this Section, configurations beyond eight threads use either two physical CPUs, HyperThreading or both, the effect of which can be negative. However, the use of such technologies is required in order to do process more threads than number of cores on a single CPU. Because of this, their overheads must be taken into account.

2. CSR is showing considerable increase in both runtime and energy consumption with the use of two physical CPUs. This will be further elaborated in Section 4.3.2.

3. Although most of the matrix groups shown benefit from the first four additional threads for both performance and energy efficiency, it is important to note that the *low_nnz_group* does not benefit from these additional threads. This group, consisting of comparably small matrices, exhibits little to no change between one and four threads, while any additional threads beyond this cause an increase in runtime and energy consumption. This is due to the low number of *nnz*s causing a too short runtime to benefit from additional threads, which, in turn, causes the overhead of thread spawning and -joining to decrease the overall performance and energy consumption at higher thread counts.

To further evaluate the comparison between CSR and CSX with regards to parallelization, Figure 4.12 contains the plot of average EDR for each of the matrix groups, in addition to the average for all matrices.

From this plot, we see that for all groups of matrices, including the group containing all matrices, the average EDR increases with the number of threads. This implies that CSX scales better with the number of threads than CSR independent of input. The only exception to the observation above is found when moving from one to two threads for the *middle_group*, consisting of matrices shown in Table 4.3. For

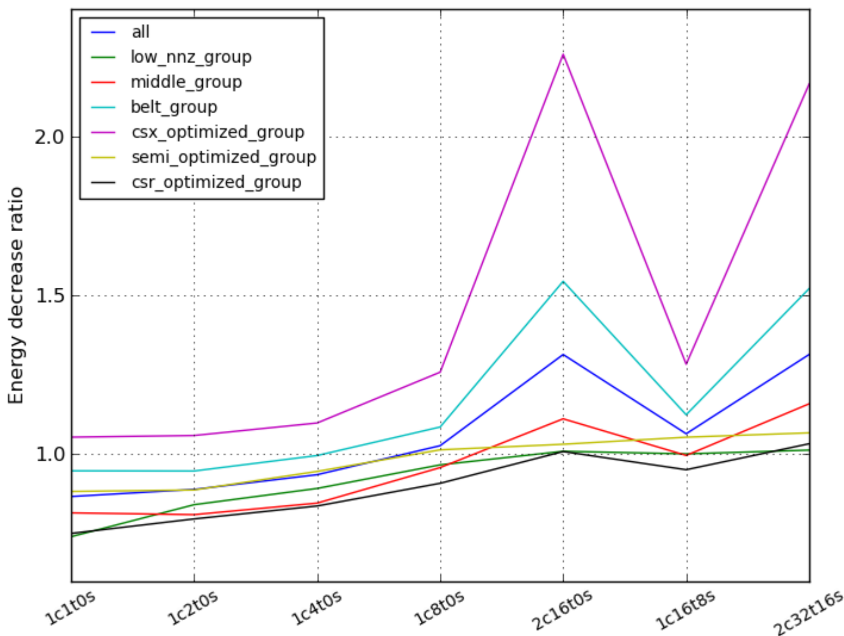


Figure 4.12: Average EDR for matrix groups

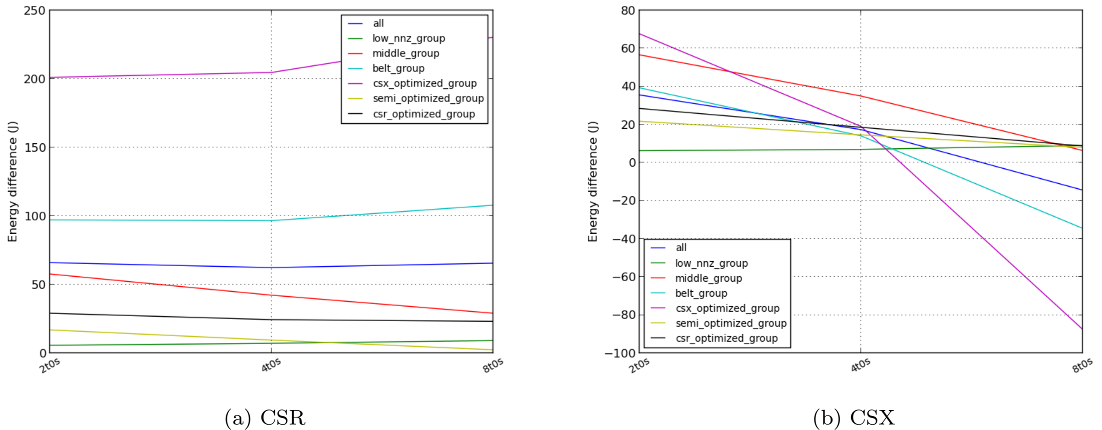
this group and these configurations, the energy consumption decrease in CSR is greater than the decrease in CSX, resulting in a lower EDR.

Another thing to note is that the matrices in general (i.e. the average of all matrices), first start to benefit from CSX at eight threads. From the discussion of Figures 4.10 and 4.11, we know that this is where CSR starts consuming additional energy compared to the optimal configurations of four cores, while this configuration provided optimal energy efficiency for CSX. From this, we can argue that when using CSX to solve CG, eight threads should be used to optimize the performance and energy efficiency gain.

4.3.2 Dual sockets

In this Section, the effect of running CSX on multiple sockets will be explored. The use of multiple sockets implies that additional physical CPUs have to be powered, hence increasing power consumption, but allowing for additional cores for processing. This increased processing capability can provide additional speedup and thus reducing the overall energy consumption of the application. For the machine setup described in Section 3.1, the effect of two sockets can be explored.

Figure 4.13 contains a plot of the average difference in CG execution energy consumption between one and two sockets for the matrix groups discussed in Section 4.2. This difference is computed by the formula $E_{2c} - E_{1c}$, resulting in positive values for configurations that show higher energy consumption for dual sockets (i.e. favor a single socket), and negative values for configurations that show lower energy consumption for dual sockets (i.e. favor dual sockets). Note the difference in y-axis values for the two plots.



(a) CSR

(b) CSX

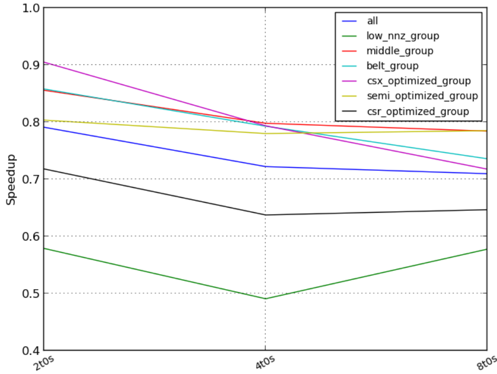
Figure 4.13: Energy difference between one and two sockets

Apparent from these plots is the fact that CSR does not favor dual sockets for any of the configurations shown. CSX, on the other hand, favors dual socket configurations with high numbers of threads for matrix groups that have been shown to benefit from CSX. Generally, we see that there is a trend towards increased benefit from dual sockets for all configurations running CSX as the number of threads increases.

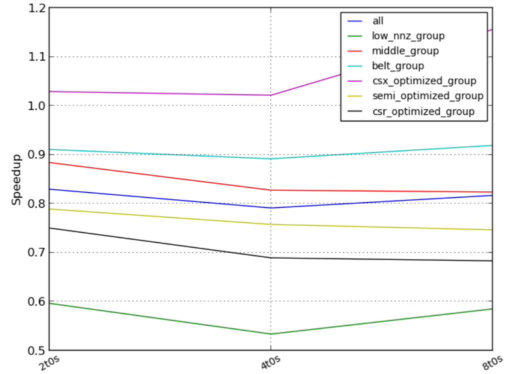
From these observations, the following implication can be made: For some matrices, CSX manages to gain enough execution performance (i.e. speedup) from the use of dual sockets to diminish the effect of the additional energy consumption caused by powering two sockets, and thus, one should be able to see significant speedup for some CSX configurations. CSR, on the other hand, should show equal performance or speeddown for all configurations.

For the most part, this is the case. Figure 4.14 shows the average group speedup of two socket configurations ($\frac{t_{1c}}{t_{2c}}$) for CSR and CSX. For CSR, the results are as one would assume, with all matrix groups showing speeddown (i.e. less than 1.0 speedup). For CSX, the results are also mostly consistent, but some major anomalies can be observed:

1. The *csx_optimized_group* shows speedup regardless of thread configurations,



(a) CSR



(b) CSX

Figure 4.14: Speedup of dual sockets

but only energy consumption decrease for eight threaded execution. However, as the eight threaded configuration is where the speedup is greatest, this can be explained with the performance increase not becoming significant enough to outweigh the energy cost of powering two CPUs until eight threads are being used. On the other hand, this can be indicative of a shift in the energy consumption comparison as the number of threads increases. This is shown to be the case in 3.

2. The *belt_group*, as well as the average for all matrices are showing slowdown for all configurations, while they obtain a decrease in energy consumption for eight threads. However, both of these matrix groups contain all the matrices of the *csx_optimized_group*. Hence, the decrease in energy consumption can be explained with the behavior of this group due to the size of its matrices affecting the average.

3. For the remaining groups, we see that while there is little to no increase in performance, they still obtain relative decrease in energy consumption as the number of threads increase. Although they never benefit from the use of dual sockets, the increase in energy consumption compared to single socket execution is minimized by the increased number of threads. This has one interesting implication: as the number of threads run by one CPU draws near to the number of cores on the CPU, its energy efficiency drops. Hence, given an execution with exactly 1.0 in speedup for all thread combinations, one can assume that the use of dual sockets will be more energy efficient than the use of a single socket for thread counts close to the limits of the single CPU.

This is also confirmed by the results for the *csx_optimized_group*, especially

apparent with the move from two to four threads. Even though the speedup stays linear, the energy difference tends towards favoring dual sockets.

For CSX, we see that large matrices (i.e. the *csx_optimized_group*) gain increased performance with the use of two sockets, regardless of thread counts. As the number of threads increase, dual socket configurations also become the most energy efficient for these matrices. For smaller matrices, on the other hand, the use of dual sockets does not prove beneficial for any configuration.

For CSR, the results show that the use of dual sockets increase both runtime and energy consumption. Therefore, all CSR executions should be run on a single socket.

4.3.3 Clock frequency

Much research has shown that the clock frequency of the CPU highly affects the energy consumption, and that throttling of the frequency can save energy for an application as a whole. Natvig *et al.* [29] described the effect of the clock frequency with Expression 4.1:

$$P_{dynamic} \sim aCV^2 f \tag{4.1}$$

where $P_{dynamic}$ is the dynamic power consumption of the CPU, a is the Activity Factor, C is the physical capacitance of the CPU, V is the supplied voltage to the CPU and f is the clock frequency of the CPU. This equation shows that the dynamic power consumption of the CPU is linearly correlated with the clock frequency.

The relation between the dynamic power consumption and the energy consumption of the CPU is by Natvig *et al.* [29] shown in Equation 4.2 and 4.3.

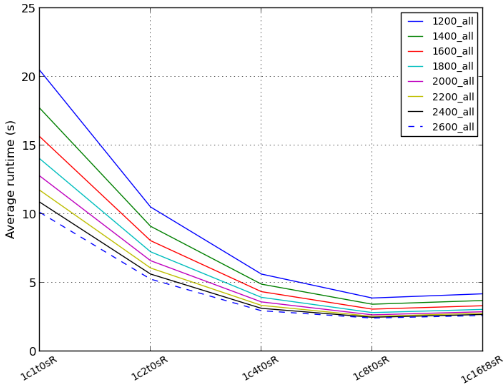
$$Power = P_{dynamic} + P_{static} \tag{4.2}$$

$$Energy = Power \times T \tag{4.3}$$

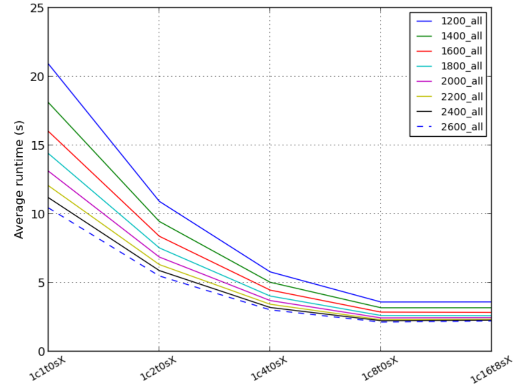
where P_{static} is the static power consumption of the CPU (i.e. the cost of having the processor powered while idling), and T is the runtime of the application.

While showing the relation between $Energy$ and $P_{dynamic}$, these two equations also show another important property of the clock frequency in relation to the energy consumption: the performance of the CPU, shown through the runtime of the application (T), is directly related to the clock frequency. This makes the frequency somewhat of a trade-off when looking at the energy consumption. On one hand, reduced clock frequency reduces the power consumption of the CPU, lowering the energy consumption. On the other hand, reduced clock frequency increases the runtime of the application, increasing the energy consumption.

This Section will thus investigate how the energy consumption is affected by throttling of the clock frequency. This is done with the applications *cpufrequtils* and *cpupower*, discussed in Section 3.2. The range of frequencies explored depends on



(a) CSR

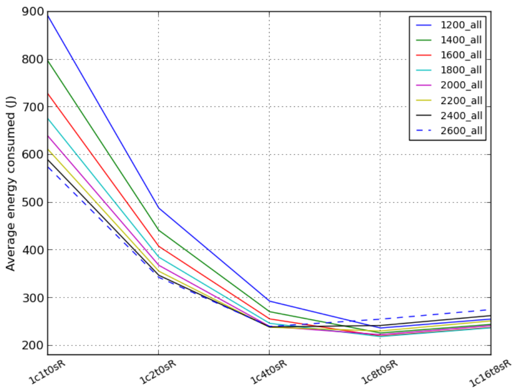


(b) CSX

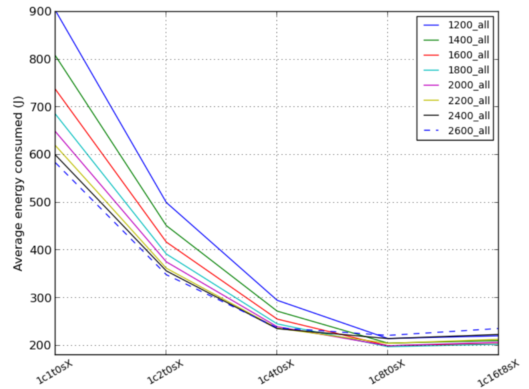
Figure 4.15: Average runtime for all matrices on differing frequencies

the hardware, and is for these experiments set to [1200, 2600]MHz with a step size of 200MHz.

Figure 4.15 contains the average runtime of all matrices for the varying frequencies. It shows, as one would expect, that higher frequencies produce lower runtimes for all configurations. This is readily explained by the clock frequency setting the speed at which the CPU can execute instructions, hence the speed at which the CPU can execute an application.



(a) CSR



(b) CSX

Figure 4.16: Average energy consumption for all matrices on differing frequencies

The energy consumption, on the other hand, shows more interesting results. As Figure 4.16 displays, the lowest frequency is not the most energy efficient for any configuration. This implies that the decrease in performance caused by the lower frequency causes the application to run for so long that the decrease in power caused by the low frequency does not result in lowered energy consumption.

When looking at the most energy efficient frequency, we see that 2600MHz (i.e. the highest frequency) is most energy efficient for low numbers of threads. However, a certain shift can be observed as the number of threads reaches eight and 16. For these two thread counts, we notice that the highest frequency becomes the least energy efficient. Generally, we notice that the frequencies in the range [2000, 2600]MHz swap their ordering and become less energy efficient for eight and 16 threads. The lower frequencies, however, retain their ordering, leaving 1800MHz the most energy efficient.

As this shows differing properties in performance and energy efficiency, the trade-off between the two is eligible for discussion. While energy can be the goal of an application study, it is not always a suited metric. This is due to energy not weighing the performance of an application highly enough, thus favoring prolonged, low power executions. Hence, more performance-weighted metrics have been proposed, such as the Energy Delay Product (EDP).

EDP is defined by Laros III *et al.* [26] in Chapter 8 of their book with Equation 4.4:

$$EDP = E \times T^w \tag{4.4}$$

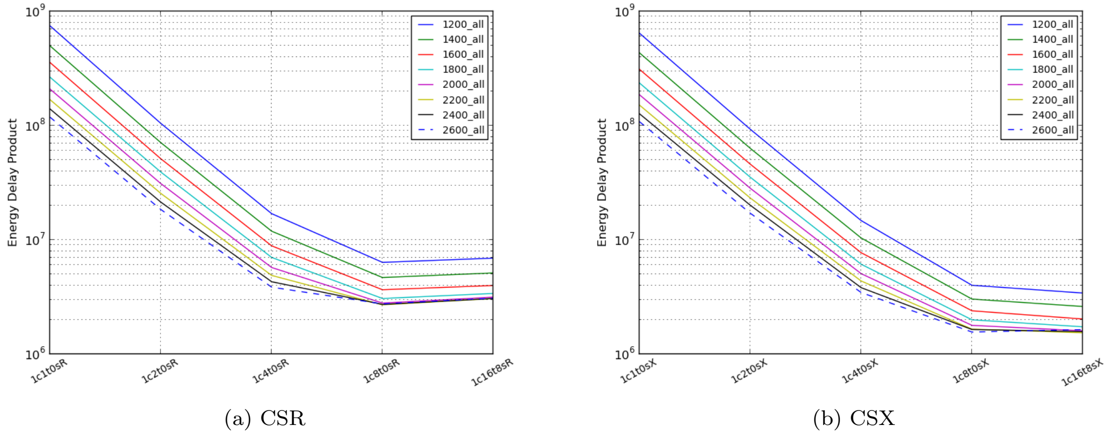
where E is the energy, T is the runtime and $w = 1$, 2 or 3. From Equation 4.4, we see that EDP weighs the energy with the runtime of the application, thus favoring configurations that are both relatively energy efficient and has high performance.

The w parameter decides how much the EDP should emphasize the performance of the application, making the decision of this parameter vital to the results. Laros III *et al.* argue that:

> For HPC, the EDP cubed equation is most appropriate due to the focus on performance. Possibly the performance factor of the metric should be weighted even higher to better represent the performance priority of HPC workloads. [26]

As the solving of SpMV is a problem found widely within scientific HPC, and that the performance of such problem solving should be relatively high, $w = 2$ will be used in this work.

Figure 4.17 shows the EDP $= E \times T^2$ plots for the differing frequencies. Note the logarithmic scale of the y-axis. For EDP, we see that higher frequencies outperform lower for all configurations. The only exception is when moving from eight to 16 threads, where the four highest frequencies ([2000, 2600]MHz) close to coincide, making them almost indistinguishable. 2200MHz does marginally outperform the others, but the difference is insignificant.



(a) CSR

(b) CSX

Figure 4.17: Average EDP $= E \times T^2$ for all matrices on differing frequencies

These observations imply that, when taking performance into the consideration, higher frequencies are favorable to lower, even though they are not as energy efficient. Because of this, we can argue that throttling of the frequency should not be done when executing CSX applications, unless energy efficiency should be obtained at the cost of performance.

4.4 Comparison of Vilje and the CARD-server

In Section 3.1, it was stated that the CARD-server was built in order to emulate a single node on the Vilje supercomputer. Hence, this Section contains discussion of similarities and, more importantly, differences between performance between the CARD-server and Vilje.

As was also stated in Section 3.1, access to the RAPL interface on Vilje was not made available for this work. Therefore, the similarities or differences in energy consumption can not be explored directly. This means that other metrics will have to be explored in order to make up an idea about whether or not results from the CARD server are representative for energy efficiency on Vilje.

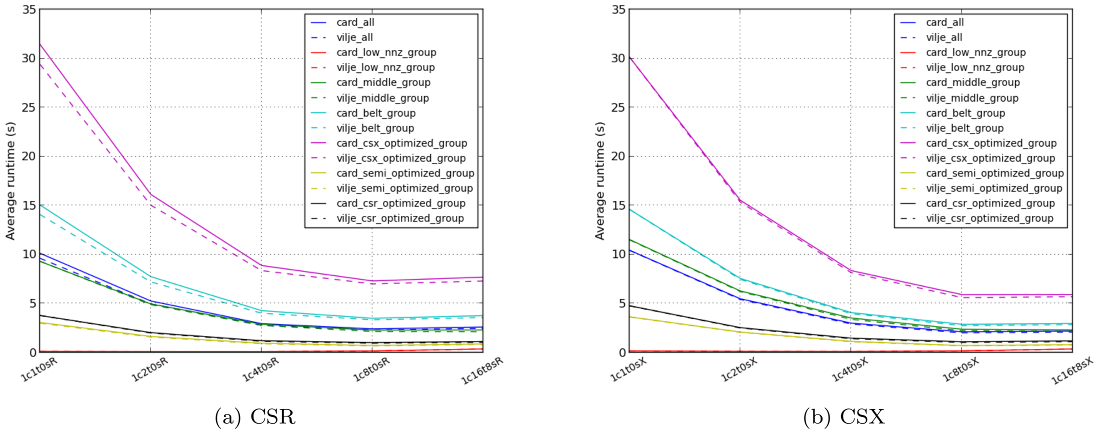
The method of using other metrics in order to make up estimates for energy consumption has been widely used within academic research prior to the inclusion of the RAPL interface in the performance counters of the CPU. One approach is the use of other performance counters to model the power and energy consumption of an application. Another approach is to look at the performance of the execution and use these results to make assumptions about the energy consumption. This approach, however, assumes correlation between the performance and energy

efficiency of an application.

For solving CG, previous work has shown that the performance and energy efficiency of both CSR and CSX seem to correlate well:

> We also see that the energy consumption and runtime for all configurations correlate strongly, both in the MSR and Yokogawa measurements. With few exceptions, we see that a reduction in runtime causes a reduction in consumed energy, and conversely, an increase in runtime causes an increase in consumed energy. [21]

Hence, a decrease in performance should indicate an increase in energy consumption, and vice versa. Therefore, the performance of the application will be used to make assumptions about the energy efficiency of Vilje versus that of the CARD-server.



(a) CSR

(b) CSX

Figure 4.18: Runtime of CARD and Vilje

Figure 4.18 shows the plots of average runtime of the matrix groups defined in Section 4.2. The solid lines represent results from the CARD-server, while the dashed lines represent results for Vilje.

Immediately apparent is the fact that Vilje outperforms the CARD-server marginally for all configurations, especially apparent for CSR executions. This can indicate one or more of three things:

1. The hardware differences between the CARD-server and Vilje affect the performance.
2. The software differences affect the execution of the application.
3. Load differences between the servers affect the results at runtime.

We can quickly disregard 1, as the hardware differences shown in Section 3.1 are limited to the amount of available RAM. Since all matrices are well within the 32GB of RAM available for a node on Vilje, the additional 32GB of RAM on the CARD-server can not make any notable difference in performance.

Option 2, on the other hand, can not be disregarded. As was shown in Section 3.2, the Linux kernel versions are radically different, with the CARD-server running on version 3.5.2-3, while Vilje is running on the older 2.6.32.59. This can highly affect performance, as the kernel is responsible for mapping the running software to the hardware. Hence, identical programs can perform differently on different kernels.

All other software, excluding GCC is identical and could therefore not affect the performance. The GCC versions, 4.6.3 and 4.6.2 for the CARD-server an Vilje respectively, could potentially alter the code slightly, but not in any major way due to the upgrade being a set of bug fixes.

Regarding option 3, it could also be responsible for the results shown. The CARD-server is a shared, single node server running a standard version of Fedora. Vilje, on the other hand, is a supercomputer with dedicated login and compute nodes. Hence, a dedicated compute node on Vilje does not have to run the same services as a single node server, reducing the strain on the node. However, the effect of this is minimal, as the activity on the CARD-server was limited during the execution of the experiments.

Moving on, we see that the graphs of Figure 4.18 retain the same general pattern for both Vilje and the CARD-server, with the slight offset discussed above. This is an indication of similar execution on both servers, with no major anomalies. Hence, we can argue, based on earlier research, that the energy profile of both servers will be similar with a slight offset.

Chapter 5

Discussion

In this Chapter, the findings of Chapter 4 will be discussed. The focus will be on unresolved issues and implications of the results presented. The Chapter is split into parts pertaining to particular observations in the results. Some of these discussions span multiple paragraphs.

The first issue to be discussed is the 1024 CG iterations used in the experiments. As was pointed out in Section 3.2.2 and during the full run comparisons in Section 4.1.2, this number of iterations does not reflect a realistic CG execution, which generally would require additional iterations in order to converge. In Section 2.2.3, it was argued that the number of iterations needed to solve CG is *O(rank)*, while Section 2.2.4 showed us that the number can be estimated from the condition number of the matrix.

To get an overview of how many iterations are realistically required to use CG to solve nLSP for the given matrix set, the condition number found in Table 3.7 can be used. From Equation 2.14, we can show that the amount of iterations (m) needed to obtain an error smaller than a threshold (t) can be given as:

$$
\begin{aligned}
2\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^m &= t \\
\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^m &= \frac{t}{2} \\
m\lg\left(\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]\right) &= \lg\left(\frac{t}{2}\right) \\
m &= \frac{\lg(\frac{t}{2})}{\lg\left(\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]\right)}
\end{aligned}
\tag{5.1}
$$

Using this equation, we can construct Table 5.1, which shows us how many CG iterations are needed in order to solve CG for two distinct error factor thresholds, namely $t_1 = 1 \times 10^{-3}$ and $t_2 = 1 \times 10^{-6}$.

Name	**condition**	**rank**	m_1	m_2
2cubes_sphere	2.9388e+09	101492	206026	393263
af_5_k101	6.4275e+08	503625	96351	183916
af_shell3	1.4403e+06	504855	4562	8707
bone010	1.2165e+09	986703	132554	253020
boneS01	4.2170e+07	127224	24680	47109
boneS10	2.9528e+08	914898	65306	124657
gyro	3.4851e+09	17361	224359	428258
LF10000	6.4743e+18	19998	9670112971	18458381769
nasa2146	4.1303e+03	2146	245	467
nasa2910	1.7650e+07	2910	15967	30477
nasa4704	1.6576e+08	4704	48930	93398
nasasrb	1.4836e+09	54870	146384	279419
olafu	2.2532e+12	16146	5704730	10889229
offshore	2.3284e+13	259789	18338507	35004674
parabolic_fem	2.1108e+05	525825	1747	3333
Pres_Poisson	3.1983e+06	14822	6797	12974
raefsky4	1.5172e+14	19779	46811934	89354959
smt	6.1260e+09	25710	297457	567788
sts4098	4.5095e+08	4098	80705	154050
thermal1	4.9625e+05	82654	2678	5111
thermal2	7.4806e+06	1228045	10395	19842

Table 5.1: Required number of CG iterations

In this Table, we see that some matrices have m_1 and m_2 smaller than their rank, implying that they can be run with fewer iterations than their rank specifies, while others have significantly larger convergence estimates than their rank, implying that the number of necessary iterations is equal to their rank. In general, we see that all matrices except *nasa2146* will have to be run for more than 1024 iterations. Seven matrices are within a factor of ten from the 1024 iterations used due to their low rank or good convergence, however, most require well beyond this.

This implies that the runs done in this work do not emphasize the time consumed by the CG execution realistically. Hence, for practically solving CG, any effect shown for the CG execution would to a greater degree affect the entire run of the application.

For the results in this work, this implies that any performance or energy efficiency gains found for the CG execution will be further emphasized in real execution, and hence, should be weighted heavier than the cost of preprocessing. This was briefly suggested in Section 4.1.2, which showed that CSX outperformed CSR for CG execution, but showed significantly worse results than CSR for the full application execution due to the cost of preprocessing.

In Section 4.1.1, plots of both the power consumption reported by the MSR and

Yokogawa were shown. It was argued that these plots showed a huge degree of correlation between the power consumption of the CPU and that of the entire system.

To further investigate this observation, the power consumption results from the MSR and Yokogawa will be correlated using the Pearson product-moment correlation coefficient (PCC). This coefficient, often denoted r, is used to show the relationship between two variables. It falls within the range[-1,1], with values close to 1 showing a *strong positive linear relationship* between the variables, and conversely, values close to -1 showing a *strong negative linear relationship* [17]. Hence, a strong correlation between the MSR and Yokogawa should show as values near 1. For further description of PCC, the reader is referred to Bluman [17] Chapter 11 or similar introductions to statistical correlation.

Table 5.2 shows the average PCC between the results of MSR and Yokogawa for each of the matrices over all available configurations. Because the Yokogawa power meter operates with an update rate of 10Hz and this causes short measurements to become inaccurate as pointed out in Section 4.1.2, a minimum runtime limit of one second was set for the PCC computation. As this makes some configurations for smaller matrices unavailable, the number #results in the Table describes the number of run configurations used to compute the average PCC.

The PCC was computed using SciPy's *pearsonr* function [6].

From the results in this Table, we can confirm the high degree of correlation between the MSR and Yokogawa power consumption results. Generally, the PCC falls between 0.7 and 1, but there are two exceptions: *nasa2146* and *thermal1*. In the case of *nasa2146*, Table 3.7 shows that this matrix is the smallest in both rank and *nnz* count, making it the most prone to measurement errors. As its PCC is based on a single run and the PCC strays far from the other matrices, this result will be interpreted as such. *thermal1*, on the other hand, can not be disregarded in the same manner. This matrix generally shows a bit lower correlation than the other matrices, but some positive correlation can still be found.

In Section 4.1.1, it was shown that an increase in power consumption for the CPU caused an additional increase in the power consumption of the entire system. Hence, it was confirmed that the energy consumption of the CPU largely dictates the energy consumption of the entire system. On the other hand, the additional increase found for the full system power indicates that other components that are not measured by the MSR dynamically influence the power of the system.

In general, the results show a great correlation between the power measured by the MSR and Yokogawa, making discussions of the MSR results relevant in the context of the energy consumption of the entire system.

When examining the matrix properties in Section 4.2, multiple groups of matrices were identified and categorized. These groups had varying properties and were identified based on their general CG performance and how well they were optimized by CSX.

Name	Average PCC	#results
2cubes_sphere	0.788234	37
af_5_k101	0.946678	37
af_shell3	0.921632	37
bone010	0.959675	37
boneS01	0.909333	37
boneS10	0.960064	37
gyro	0.805220	25
LF10000	0.998960	1
nasa2146	-0.142734	1
nasa2910	0.760364	5
nasa4704	0.791872	2
nasasrb	0.857245	37
olafu	0.856492	15
offshore	0.900177	37
parabolic_fem	0.876333	37
Pres_Poisson	0.754447	12
raefsky4	0.711928	17
smt	0.759281	37
sts4098	0.917723	2
thermal1	0.644362	28
thermal2	0.920648	37

Table 5.2: Average Pearson Product-Moment Correlation Coefficient

Two main parameters were found to determine the performance for the matrices: size in number of *nnz*s and structure. It was found that very small matrices performed poorly for CG execution in general and that matrices smaller than the LLC were not optimized significantly by CSX. For structure, it was found that elaborate matrix patterns with a lot of values far from the diagonal perform poorer than more regular matrices for CG execution, and that clustered, diagonal centered matrices optimized the performance benefit from CSX.

A low number of *nnz*s will, as one would expect, result in a reduced runtime of the application. This causes the startup and overhead of the application to take up a larger part of the overall energy consumption. Because of this, the high energy/*nnz* for the small matrices is as one would expect. Larger matrices, on the other hand, cause the application to spend more time in processing, limiting the effect of startup and general overhead and causing an improvement in the energy/*nnz* ratio.

Regarding the effect of matrix size on the CSX performance, the results are as expected and by design. Kourtis *et al.* writes this in their presentation of CSX:

> Our previous work [...] has identified the memory subsystem as the main performance bottleneck of the SpMV kernel. Obviously, this problem becomes more severe in a multithreaded environment, where multiple

> processing cores access the main memory. An approach for alleviating this problem is the reduction of the data volume accessed during the execution of the kernel (working set). [25]

Hence, as CSX optimizes by reducing data volume of the working set of the matrix during execution, the optimization increases in performance gain as the size of the matrix increases. This increase becomes even more significant when the size of the matrix surpasses the cache size of the CPU. When this happens, the number of memory accesses significantly increases, causing the CSX optimization to become even more beneficial.

This is further emphasized in Figure 5.1, which shows the amount of L3 cache misses normalized with the size of the matrix (i.e. *nnz*s). Apparent from this Figure is that the matrices which size surpasses the LLC of the CPU (*af_5_101*, *af_shell3*, *bone010*, *boneS01*, *boneS10* and *thermal2*), all generate relatively large amounts of L3 misses. This makes them well suited for CSX optimization, and they are therefore mostly found in the *csx_optimized_group*.

However, *thermal2*, which generates the most misses is not found in this group. In addition, we see that several matrices smaller than the LLC cache size, namely *parabolic_fem*, *offshore* and to a lesser degree *smt*, also generate relatively large amounts of L3 misses. This is attributed to their structure.

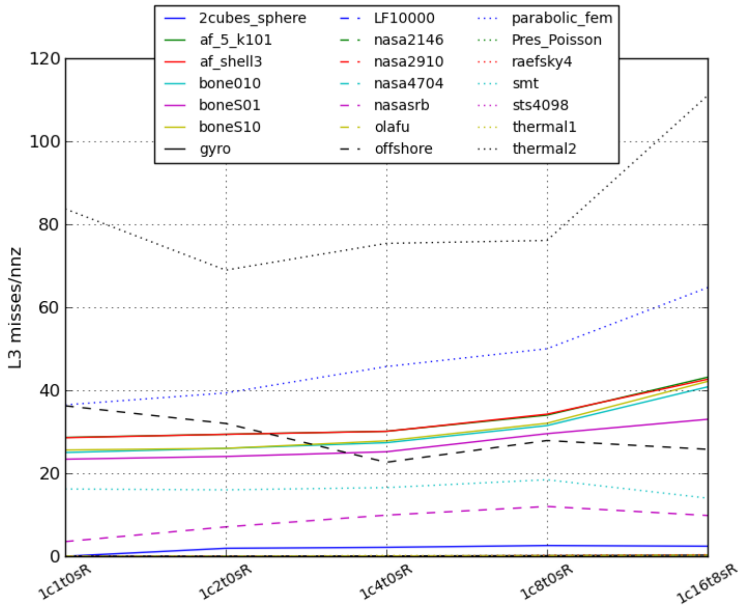
Structurally, it was observed that elaborate matrix patterns generally perform poorly for CG in general, and are not well optimized by CSX. Hence the reason the matrices mentioned above are not part of the *csx_optimized_group*. Their spread values make them hard to efficiently access by the SpMV kernel, thus limiting the performance of the CG solver. Their structure also limits the optimization obtainable by the delta encoding done by CSX, which works by identifying clustered values and compressing them into units. This makes matrices with a large number of spread values generate a lot of small units, causing the overhead to decrease the general performance of the optimization.

CSX should therefore be used on large matrices with a high amount of clustered values. Specifically, CSX should be used on matrices that are larger than the size of the LLC of the machine.

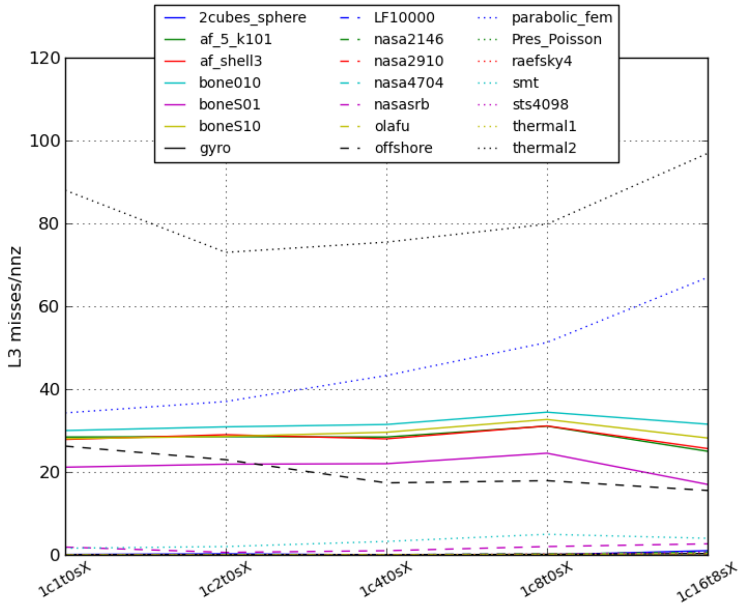
Regarding the parallelization properties presented in Section 4.3.1, we have already established that the use of multiple threads significantly reduces the runtime and energy consumption of the application up to a certain point. We have also established that CSX parallelizes better than CSR, resulting in higher performance and energy efficiency increase.

It also found that the optimal runtime configuration generally was located at a higher thread number than the optimal energy efficiency configuration, and that similarly, CSX was optimized for both these categories at a higher thread count than CSR.

The reason for the latter observation can be found in the discussion of matrix group



(a) CSR



(b) CSX

Figure 5.1: L3 misses per nnz

categorization above. As was noted by Bell *et al.* [14] and Kourtis *et al.* [25], the main bottleneck for solving SpMV is the memory subsystem. This bottleneck becomes even more significant as the number of threads increase, causing the rate of data being streamed to the cores to increase due to the additional processing capability of the CPU. As the number of threads keep increasing, the rate of memory accesses causes the memory bottleneck to limit the performance. This is shown in Figure 5.2, which shows the number of L3 misses per second. As CSX is designed to decrease the cost of each of these memory accesses, it can support more threads than CSR without the memory bottleneck becoming significant.

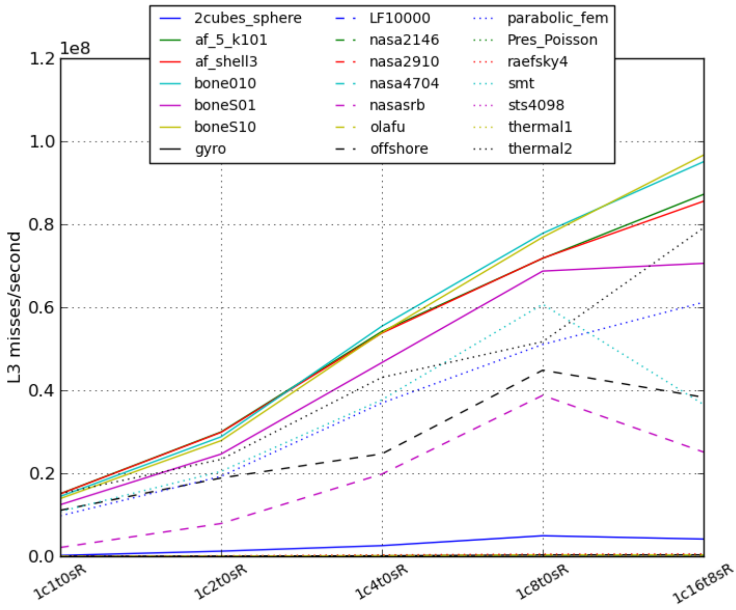
Regarding the first observation of runtime decreasing even though energy consumption is increasing for high numbers of threads, it was mentioned in Section 4.3.2 that the energy efficiency of a CPU seems to diminish as the number of threads closes in on the number of cores on the CPU. Even though there is still performance increase to be found, this decreased energy efficiency causes the application to consume additional energy. Hence, to maximize the energy efficiency, one should not necessarily maximize the number of threads.

When examining the effect of using two sockets (i.e. two physical CPUs) in Section 4.3.2, it was found that CSR had increased runtime and energy consumption when using dual sockets regardless of thread configuration, while CSX showed decreased runtime for large matrices for all thread configuration and decreased energy consumption large matrices for high numbers of threads.

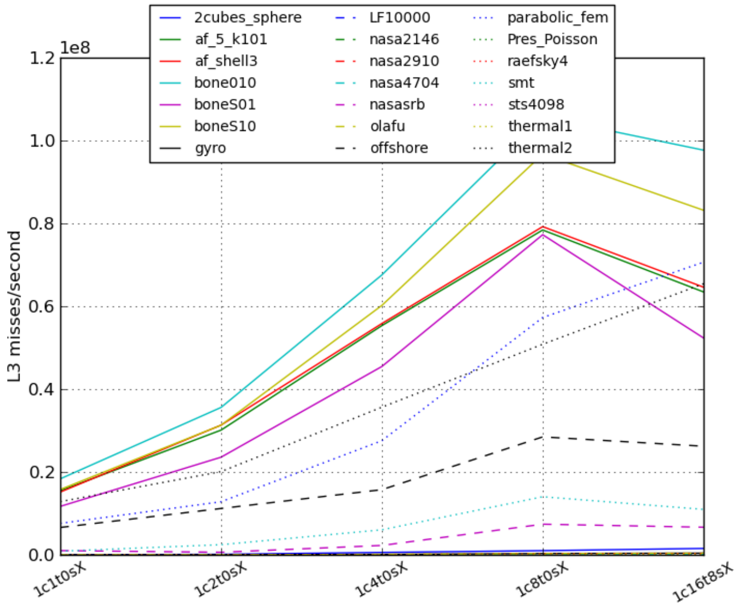
This observation implies that the use of dual sockets causes the memory bottleneck to increase. This does not benefit matrices or configurations that are not optimized by CSX, such as small matrices or low numbers of threads. Given a configuration that is shown to generally benefit from the use of CSX, however, the use of dual sockets causes additional increase in performance and energy efficiency.

As was mentioned, the reason for this is an increase in the memory bottleneck caused by the use of dual sockets. As the matrix is split between the two CPUs, they have to communicate through the main memory, causing additional accesses. There is also some synchronization needed between the processors at each CG iteration, adding additional overhead to the execution. On the other hand, each of the processors only have to execute half the number of threads, causing increased energy efficiency due to the CPUs low energy efficiency at high numbers of threads discussed above. This also causes the amount of required LLC for each of the CPUs to decrease, as each of them only have to process half the matrix. This can be seen in Figure 5.3, which shows the L3 miss rate for single socket and dual socket configurations.

For large matrices, the overhead of the dual sockets is marginalized by the runtime of the CG execution, causing the increased CPU energy efficiency at high numbers of threads and the reduced number of L3 cache misses to increase performance and overall energy efficiency. The CSX optimization also minimizes the cost of the memory accesses required to ensure synchronization and communication between the CPUs.

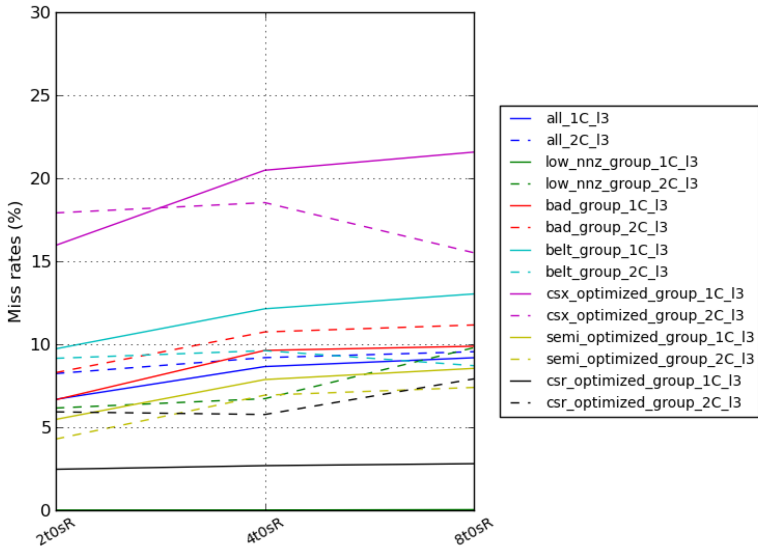


(a) CSR

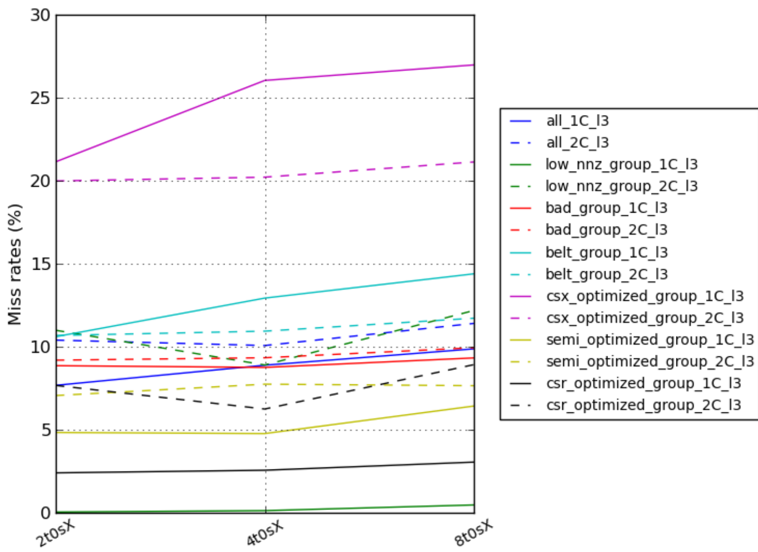


(b) CSX

Figure 5.2: L3 misses per second



(a) CSR



(b) CSX

Figure 5.3: L3 miss rates

For smaller matrices that fit in the LLC cache, the reduced working set of each CPU does not benefit the execution. On the contrary, the memory accesses required for synchronization and communication causes an overall increase in the number of memory accesses, reducing the performance and energy efficiency of the application. For CSR, which does not optimize the memory accesses, all increase in these cause reduced performance and energy efficiency.

Therefore, dual sockets should only be used for CSX with matrices that are larger than the LLC of the CPUs and for thread configurations near the limits of each CPU.

Regarding throttling of the CPU frequency presented in Section 4.3.3, it was shown that throttling had some effect with regards to energy consumption, but also increased the runtime of the application accordingly. It was also shown that when weighing the energy consumption with the runtime using the EDP formula of Laros III *et al.* [26] with a weight exponent of two ($w = 2$), the optimal frequencies were generally found near the maximum frequency of the CPU.

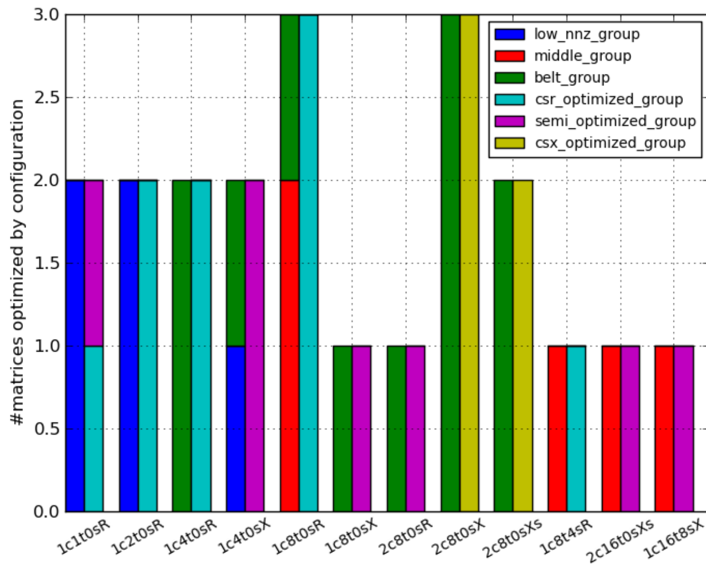
As these results show, the lowest frequency possible was never the most energy efficient. The optimal configurations were found at 2600MHz (i.e. the maximum frequency) for thread counts of four and lower, while 1800MHz became the optimal frequency for eight threads or more. What this implies is that upon adding the four additional threads, the extra performance increase does not outweigh the increase in power caused by the additional cores. This can be explained by the increased rate at which data has to be streamed to the cores increasing with the number of threads, as shown in Figure 5.2. This causes strain on the cache hierarchy, increasing the wait time per core, and thus increasing the amount of energy burned while waiting for cache misses as the frequency increases.

For this reason, we can argue that throttling of the CPU frequency should be considered only when running a high number of threads. It should, similarly, only be considered when trying to minimize the energy consumption of the execution at the cost of performance, as the weighted results show the best trade-offs near the maximum frequency.

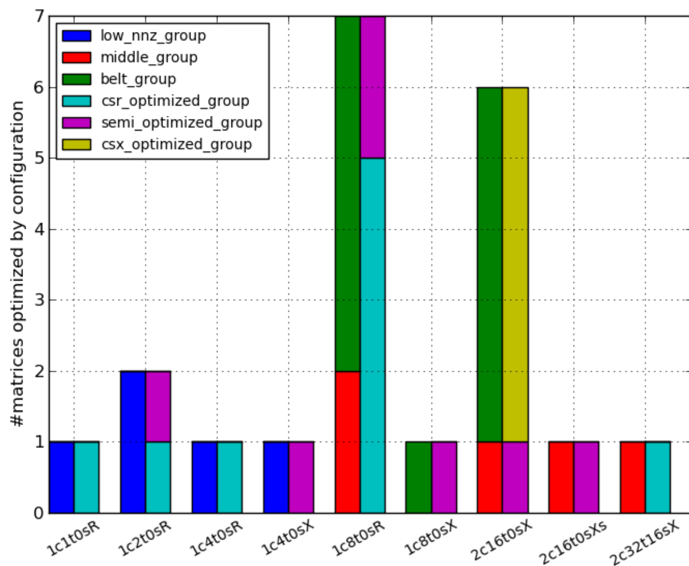
In order to confirm and sum up the discussion up to this point, Figure 5.4 contains a plot of the optimal configurations for the CG execution phase for the matrices. Hence, the cost of the preprocessing discussed earlier in this Chapter is not reflected in this plot. The plot also shows how the matrix groups identified in Section 4.2 are distributed across these optimal configurations.

Starting with the energy efficiency plot of Figure 5.4(a), we see that the matrices are distributed among several different configurations, with the bulk of the matrices being optimized by between four and eight threads. The following observations can be made from the plot:

1. The matrices optimized at lower thread counts than four are all part of the *low_nnz_group*, showing that very small matrices should not be run with high degrees of parallelization.



(a) Energy Efficiency



(b) Performance

Figure 5.4: Optimal configurations for the CG execution

2. The *csr_optimized_group* is optimized by CSR configurations, and similarly, the *csx_optimized_group* is optimized by CSX configurations, confirming their categorizations. The *semi_optimized_group*, on the other hand, are optimized by both CSR and CSX.

3. All matrices optimized by 16 threaded configurations are optimized by CSX. This is due to the higher tolerance for L3 cache miss rate provided by CSX discussed above.

4. Interestingly, some matrices are optimized by runs of CSX with statistical sampling, which one would think would perform worse than the full preprocessing of CSX. Namely, these are the *bone010* and *boneS10* matrices, which are optimized by the *2c8t0sXs* configuration, and *parabolic_fem*, which is optimized by the *2c16t0sXs* configuration.

   This implies that the patterns identified by the statistical sampling for these matrices are better than the patterns identified by the full preprocessing of the matrix. It shows that the significantly reduced cost of preprocessing offered by statistical sampling does not reduce the quality of execution significantly. On the contrary, it does in some cases increase the performance of the execution.

5. Many of the matrices optimized by CSX are optimized for dual socket configurations, confirming the observation that CSX manages to overcome the additional memory accesses needed for inter-socket communication.

   More interestingly, one matrix is optimized for dual socket configuration while running CSR, contradictory to the observations done in Section 4.3.2. The matrix in question is the *smt* matrix, which is optimized for the *2c8t0sR* configuration. As was noted, CSR generally did not benefit from the use of dual sockets. However, Figure 4.13(a) shows that the *semi_optimized_group*, of which *smt* is a member, on average almost benefits from the use of dual sockets for CSR.

   The *semi_optimized_group* was categorized by its matrices being smaller than the LLC with elaborate structures. Hence, they do not significantly benefit from the memory optimization of CSX due to their size, but also perform poorly for CSR due to their structure. This is the reason why these matrices perform pretty well with dual sockets for CSR. Their poor single core CSR performance is worsened as the number of threads increase, due to the lowered energy efficiency of a CPU as the number of threads reaches the limits for the CPU. On the other hand, the overhead of CSX causes its performance to drop due to the size of the matrix making it fit in the LLC. For these reasons, the use of dual sockets in CSR becomes optimal, but only for one of the matrices and only marginally better than its single core execution.

Moving on to the performance plot of Figure 5.4(b), we see that the distribution of matrices are centered around two configurations, namely *1c8t0sR* and *2c16t0sX*. This is again consistent with the results and discussion of parallelization, showing that CSX parallelizes better than CSR and thus manages to gain performance at

higher thread counts.

As with the energy efficiency, the lowest thread count optimizations happen for matrices of the *low_nnz_group*, due to their small size.

Interestingly, one of the matrices of the *csr_optimized_group*, namely *thermal2*, is optimized by a CSX configuration at the highest possible number of threads. This matrix is larger than the LLC cache, implying that it should benefit from CSX, but its structure makes it perform better for CSR at lower thread counts. Because of this, we can argue that the size of the matrix becomes even more significant than the matrix structure as the number of threads increases beyond 16.

From these results, three major general implications can be drawn:

1. Matrices that are shown to benefit from CSX should be run with dual sockets where available, However, one should not maximize the number of threads available, as the energy efficiency of the CPU is shown to decrease as the number of threads reach the maximum of the CPU. Hence, even though the performance when running eight threads per core might be higher than when running four, the consumed energy might increase in spite of the lower runtime.

2. As CSR generally does not benefit from the use of dual sockets, one should keep its executions on a single core. The above discussion of energy efficiency of a CPU with the number of threads also holds here, so one should not necessarily maximize the number of available threads when trying to minimize energy consumption.

3. One should use statistical sampling if available when running CSX. It is shown to have significantly decreased preprocessing cost when compared to full preprocessing, without significant loss of SpMV execution performance. In some cases, it is even shown to surpass the performance of full preprocessing.

However, as was stated in Section 4.1.1, statistical sampling could not be used for the entire matrix set of this work. Hence, its full potential could not be explored thoroughly. On the other hand, the preliminary results of the limited matrix set that was able to run with statistical sampling is showing great potential for this technique. The SpMV execution performance is shown to be close to or equally good as full preprocessing, and its cost is significantly reduced in comparison.

Section 4.1.2 showed that when compared to CSR, the full application energy consumption at 1024 CG iterations of the two techniques where almost indistinguishable at high numbers of threads, while the SpMV performance became significantly higher than that of CSR with the high number of threads.

As has been shown in the beginning of this Chapter, 1024 iterations do not represent a realistic CG execution, as the number of iterations needed generally is higher. Hence, one can expect CSX with statistical sampling to outperform CSR for full application energy consumption upon using it to solve CG problems.

In Section 4.4, it was found that the performance of a compute node at Vilje was performing slightly better than the CARD-server. However, the graphs in Figure 4.18 showed that the general performance profile of the different matrix groups were similar on both servers. Upon applying the Pearson product-moment correlation coefficient presented earlier in this Chapter to these graphs, the results of Table 5.3 are found.

Group Name	**CSR PCC**	**CSX PCC**
all	0.999986	0.999999
low_nnz_group	0.999914	0.999492
middle_group	0.999986	0.999989
belt_group	0.999971	0.999995
csx_optimized_group	0.999975	0.999993
semi_optimized_group	0.999984	0.999976
csr_optimized_group	0.999983	0.999990

Table 5.3: Pearson Product-Moment Correlation Coefficient for CARD and Vilje comparison

This Table shows significant correlation between the graphs, implying that executions on the CARD-server are indicative of executions on a compute node of Vilje.

However, these results were but performance results for the different machines, caused by the unavailability of the MSR register on the Vilje super computer nodes. Because of this, energy comparisons could not be made. On the other hand, it has already been established that the runtime and energy consumption of the application are correlated. Some discrepancies have been identified, such as reduced energy efficiency at thread counts near the limits of the CPU, but the overall correlation has been found to be significant. Based on this observation, and the high correlation in runtime between a compute node on Vilje and the CARD-server, similar results in energy consumption between the two machines can be expected, making results on the CARD-server indicative of results that can be expected on the Vilje supercomputer. One must note, however, that this can not be confirmed until energy measurements are made available on Vilje.

One major thing to note is that all runs on the Vilje supercomputer are done at a single compute node. This is due to one of the design limitations of the CSX framework, being designed around shared memory access.

As was presented in Section 2.4.2, the framework obtains parallelization by splitting the *nnz*s of the matrices among the running threads. This is done by letting each thread fetch a limited part of the matrix from the main memory before starting the execution.

In order to make CSX span multiple nodes, one would have to redesign the framework to utilize other parallelization techniques, such as Message Passing Interface

(MPI) [8] or OmpSs [10]. This would make several compute nodes able do processing rather than a single node, resulting in increased parallelization at the cost of the inter-node communication overhead.

However, this is not necessarily beneficial. As we have established throughout this work, SpMV is a memory intensive operation, causing the memory bottleneck to limit the performance of execution. CSX is designed to remedy this, by decreasing the memory data volume and thus limiting the effect of the bottleneck. While adding additional compute nodes to the problem would multiply the computational capacity of the machine, the cost of accessing the full data set would increase drastically as its parts would be spread among the compute nodes.

Due to this property of SpMV and based on the results found in this work, it is not likely that the use of multiple nodes would be beneficial to the performance of its execution because of the additional cost of inter-node memory access. On the other hand, the use of such techniques could be beneficial for sufficiently large matrices (i.e. matrices that surpass the size of the main memory causing disk access to become a factor), but as no such matrices have been examined in this context, any discussion on this would be baseless and is thus omitted.

Chapter 6

Conclusion and Further Work

This Chapter contains the conclusions made based on the results of Chapter 4 and discussions of Chapter 5 in addition to suggestions regarding what should be looked into in a continuation of this study.

6.1 Conclusion

As has been established throughout this work, CSX is showing great performance and energy efficiency potential when compared to simpler, well tested methods for solving SpMV. However, in order to fully benefit from the optimizations provided by CSX, one must have a well suited problem and run it at a well suited configuration.

For the matrix set, two main parameters have been identified to influence the performance and energy efficiency gain of CSX, namely the matrix size and the structure.

The size of the matrices in amount of nnzs has been shown to correlate well with the expected energy efficiency gain of CSX. Specifically, a threshold has been identified at the size of the LLC, with matrices larger than this threshold generally benefiting from CSX, and matrices smaller than this threshold becoming dependent on other factors.

Hence, CSX should be used for matrices larger than the LLC of the targeted architecture. The additional main memory accesses caused by the size of these matrices greatly benefit from the data volume optimizations of CSX. For smaller matrices which fit in the LLC of the targeted architecture, the reduced number of main memory accesses minimizes the effect of the CSX optimizations, causing a trade-off between memory access gain and execution overhead.

Structurally, it has been found that the degree of clustering in the matrix values influence the performance of CSX. Matrices with a high amount of clustered values, such as diagonal-based matrices, are generally well optimized by CSX, while matrices with more dispersed values, especially matrices with a high number of values far from the diagonal, do not benefit significantly from these optimizations and therefore favor less complicated storage schemes due to the overhead.

One should therefore consider the matrix set when using CSX, as it will not be beneficial for performance or energy efficiency for certain configurations. For a well suited matrix set, however, CSX will increase both the performance and energy efficiency of the SpMV execution.

Regarding platform properties and amount of parallelization, it has been shown that while some parallelization is required to benefit from CSX, one should not necessarily run with as high degree of parallelization as possible. For the given architecture in this work, it has been found that the best trade-offs are generally found at eight threads for energy efficiency and 16 threads for performance. However, this is relative to the number of physical cores on each CPU.

The use of dual sockets has been shown to increase both the performance and energy efficiency of CSX executions, contrary to similar executions for CSR. For this reason, the use of multisocket platforms will be beneficial to the execution of CSX, and should therefore be favored.

Throttling of the frequency has been shown to have some effect on the energy efficiency of the execution, but it is not linearly correlated. Hence, due to the decrease in performance, throttling should be used sparingly and only when trying to achieve decrease in energy consumption at all costs. When trying to achieve a balance between energy efficiency and performance, running at close to maximum frequency is showing the best results.

With regards to the CARD-servers ability to emulate the execution and results, it has been shown that the execution on the two machines is similar, with a compute node on Vilje generally performing slightly better compared to the CARD-server in performance. Based on this, and the correlation between runtime and energy consumption shown throughout this work, one can expect similar energy efficiency results for the two machines with energy efficiency slightly favoring Vilje. Hence, the CARD-server is shown to well emulate the performance and energy efficiency one can expect to find for a single node on the Vilje supercomputer.

To conclude, this work has identified properties both in the input matrices and in the target platform that can be used to maximize the energy efficiency of CSX. It has shown that large matrices with a diagonal structure are best suited for CSX optimization, and that matrices with spread clusters of values and matrices that are smaller than the LLC of the target architectures generally are not as beneficial. It has also been shown that the optimal parallelization for CSX generally is found at eight threads running on two physical CPUs, that this most likely is related to the number of cores on each CPU, and that throttling of the frequency has a small,

but significant effect on the energy efficiency of the application.

CSX is showing great potential for increasing the energy efficiency of SpMV execution, but it is not well suited in all cases. Hence, the significant problem and platform properties identified in this work should be evaluated when deciding whether or not to use CSX for the given problem.

6.2 Further Work

In this Section, ideas on how to augment this work and proceed with further energy studies of CSX are presented.

The results in Section 4.1.2 show that while CSX in many cases obtain significant increases in the energy efficiency of the CG execution, the cost of preprocessing keeps it from surpassing CSR in overall energy efficiency for 1024 CG iterations. However, in Chapter 5, a method of estimating the required number of CG iterations was presented. Based on this, a continuation of this work should include a discussion of the trade-off between preprocessing cost and CG execution gain. This especially holds for CSX running with statistical sampling, which is showing a potential increase in this trade-off.

One of the major drawbacks of this work is the unavailability of statistical sampling executions on several matrices. This issue has been reported to the CSlab at the NTUA, the creators of CSX, and a fix could therefore possibly be available during a continued study of CSX. This work has to some degree shown the potential benefits of the statistical sampling, and this should therefore be further studied, if available.

From the results, we have seen that the energy efficiency of the CPUs drop as the number of threads increases from four to eight. This, however, is correlated with the number of cores on the CPUs, and hence, the study of this effect is very limited due to the hardware used in the experiments. To thoroughly study this effect, one should make sure to run executions on several different architectures with various numbers of cores per CPU. With this, one should be able to find if this effect is indeed correlated with the number of cores, or whether other factors not apparent from the results in this work are contributing to the results.

The matrix set chosen for this work is generally evenly distributed among the parameter set, but one drawback can be found: most of the matrices that are larger than the LLC of the machine are simple in structure. The only matrix this does not apply to is *thermal2*, which is both larger than the LLC and has an elaborate structure. However, this limits the study of large, structurally elaborate matrices, which could potentially provide additional insight into matrices and their parameters influence on the energy efficiency of CSX.

This limit of the matrix set is caused by the limitations of CG and available matrices of the University of Florida Sparse Matrix Collection [11], and thus, a continuation

of this work should strive to obtain additional sources of matrices in order to improve the studied matrix set.

As was pointed out in Section 3.1, access to the MSR on the nodes of the Vilje supercomputer was not made available in the course of this work. Therefore, all discussions of the differences in execution between the CARD-server and Vilje was based on the performance result of both machines and the correlation found between runtime and energy consumption. To confirm these results, one will have to perform energy studies on both machines, and thus, this should be prioritized upon further evaluation of equalities and differences between the CARD-server and Vilje.

In Chapter 5, methods of making CSX able to span multiple compute nodes were briefly discussed. As was pointed out, this ability will possibly decrease the performance of the execution for matrices of the size studied in this work, but could potentially be beneficial for very large matrices (larger than the main memory of a single node). Making the changes to CSX in order to support this would require a major rewrite of the framework, but could potentially be done to study the effect of multiple nodes or multiple devices (such as GPUs).

Bibliography

[1] About Vilje. https://www.hpc.ntnu.no/display/hpc/About+Vilje, Apr 2013.

[2] cspy, a MATLAB function in the CSparse package. http://www.cise.ufl.edu/research/sparse/CSparse/CSparse/MATLAB/CSparse/cspy.m, Apr 2013.

[3] CSX GitHub. https://github.com/cslab-ntua/csx, Apr 2013.

[4] Intel® Energy Checker SDK. http://software.intel.com/en-us/articles/intel-energy-checker-sdk, Apr 2013.

[5] Intel® Energy Checker SDK Device Driver Kit User Guide. http://software.intel.com/sites/default/files/m/d/4/1/d/8/Intel_28R_29_Energy_Checker_SDK--Device_Kit_User_Guide--2010_12_15.pdf, Apr 2013.

[6] Module SciPy.stats.stats. http://www.scipy.org/doc/api_docs/SciPy.stats.stats.html#pearsonr, May 2013.

[7] Om Meteorologisk institutt - met.no. http://met.no/Om_oss/Om_Meteorologisk_institutt/, May 2013.

[8] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/, May 2013.

[9] [Ptools-perfapi] Using PAPI with PThreads. http://lists.eecs.utk.edu/pipermail/ptools-perfapi/2011-January/001898.html, May 2013.

[10] The OmpSs Programming Model — Programming Models @ BS. https://pm.bsc.es/ompss, May 2013.

[11] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/, Jan 2013.

[12] Threads - PAPIDocs. http://icl.cs.utk.edu/projects/papi/wiki/Threads, May 2013.

[13] Top500 List - June 2012. `http://www.top500.org/list/2012/06/100/`, Apr 2013.

[14] Nathan Bell and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Corporation, 2008.

[15] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. *Decomposable and Responsive Power Models for Multicore Processors using Performance Counters*. Proceedings of the 24th ACM International Conference on Supercomputing, 2010.

[16] W. L. Bircher, M. Valluri, J. Law, and L. K. John. *Runtime Identification of Microprocessor Energy Saving Opportunities*. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.

[17] Allan G. Bluman. *Elementary Statistics: A Step by Step Approach*. McGraw-Hill, third edition, 1998.

[18] Edwin K. P. Chong and Stanislaw H. Zak. *An Introduction to Optimization*. John Wiley & Sons Inc., second edition, 2001.

[19] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. *Dark Silicon and the End of Multicore Scaling*. IEEE, 2011.

[20] Wu-chun Feng and Kirk W. Cameron. *The Green500 List: Encouraging Sustainable Supercomputing*. IEEE, 2007.

[21] Lars-Ivar Hesselberg Simonsen. *Energy efficiency of CSX: A preliminary study of the energy trade-offs when using compression to optimize Sparse Matrix Vector Multiplication*. 2012.

[22] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. *Measuring Energy Consumption for Short Code Paths Using RAPL*. GREENMETRICS, 2012.

[23] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. *Exploring the Performance-Energy Tradeoffs in Sparse Matrix-Vector Multiplication*. National Technical University of Athens, 2011.

[24] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. *Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression*. Proceedings of the 2008 conference on Computing frontiers, 2008.

[25] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. *CSX: An Extended Compression Format for SpMV on Shared Memory Systems*. Principles and Practice of Parallel Programming (PPoPP), 2011.

[26] James H. Laros III, Kevin Pedretti, Suzanne M. Kelly, Wei Shu, Kurt Ferreira, John Van Dyke, and Courtenay Vaughan. *Energy-Efficient High Performance Computing: Measurement and Tuning*. Springer, 2013.

[27] Hallgeir Lien. *Case Studies in Multi-core Energy Efficiency of Task Based Programs.* Norwegian University of Science and Technology, 2012.

[28] Jan Christian Meyer, Lasse Natvig, Vasileios Karakasis, Dimitris Siakavaras, and Konstantinos Nikas. *Energy-efficient Sparse Matrix Auto-tuning with CSX.* Proceedings of the 27th IEEE Intl. Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013.

[29] Lasse Natvig and Alexandru C. Iordan. *Green Computing: Saving Energy by Throttling, Simplicity and Parallelization.* CEPIS UPGRADE, 2011.

[30] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, Christos Kozyrakis, and Justin Meza. *Models and Metrics to Enable Energy-Efficiency Optimizations.* IEEE, 2007.

[31] Youcef Saad. *SPARSKIT: a basic tool kit for sparse matrix computations.* 1994.

[32] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Second with corrections edition, 2000.

[33] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* Carnegie Mellon University, 1994.

[34] Karan Singh, Major Bhadauria, and Sally A. McKee. *Real Time Power Estimation and Thread Scheduling via Performance Counters.* ACM SIGARCH Computer Architecture News, 2009.

[35] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. *OSKI: A Library of Automatically Tuned Sparse Matrix Kernels.* SciDAC 2005 Proceedings, 2005.

[36] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. *Optimization of sparse matrix–vector multiplication on emerging multicore platforms.* Elsevier, 2008.

Appendices

Appendix A

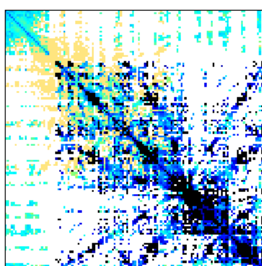
Matrix Structures

In this Appendix, the structure of each of the matrices used in the experiments are presented. The plots are provided alongside the matrices in the University of Florida Sparse Matrix Collection [11].

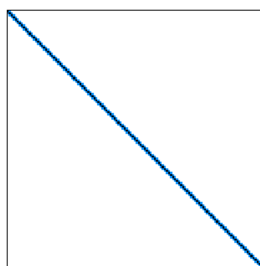
The plots were generated using a script called *cspy.m*, which is part of the CSparse MATLAB package [2]. The colors of the plots are given as follows:

> Zero entries are white. Entries with tiny absolute value are light orange. Entries with large magnitude are black. Entries in the midrange (the median of the log10 of the nonzero values, +/- one standard deviation) range from light green to deep blue. [2]

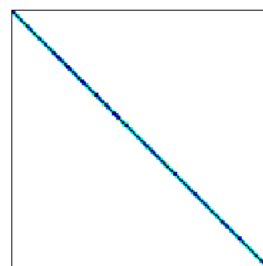
The plots are static in size and number of pixels, so they do not properly display the rank of the matrices. Because of this, one should refer to the list of matrices in Section 3.3 while examining these plots.



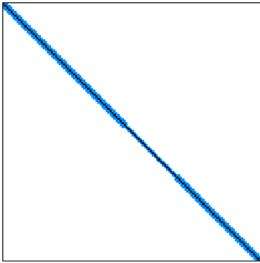
(a) 2cubes_sphere



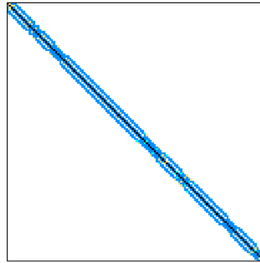
(b) af_5_k101



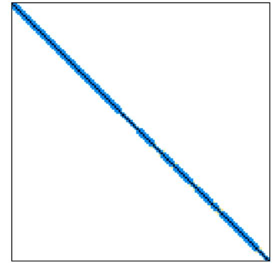
(c) af_shell3



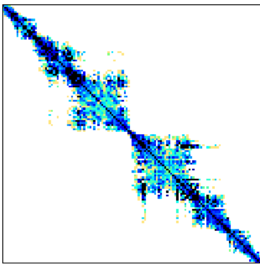
(d) bone010



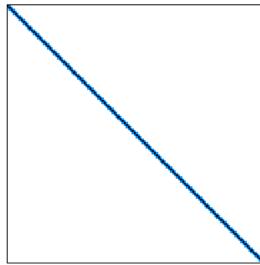
(e) boneS01



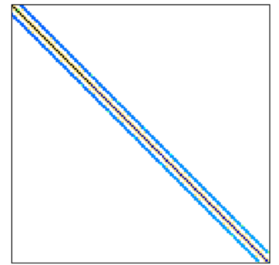
(f) boneS10



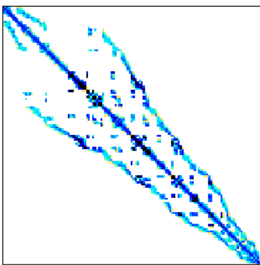
(g) gyro



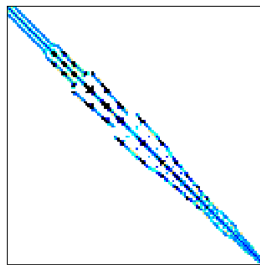
(h) LF10000



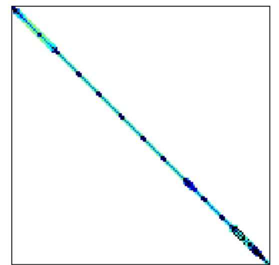
(i) nasa2146



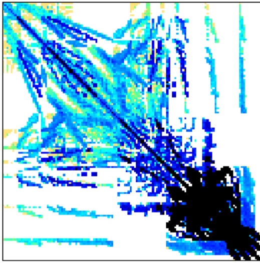
(j) nasa2910



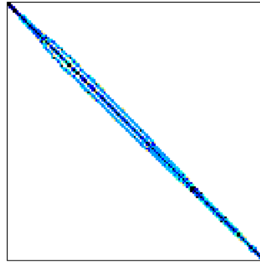
(k) nasa4704



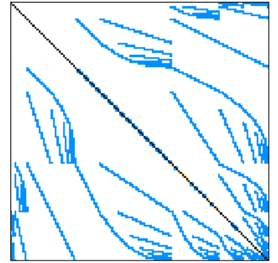
(l) nasasrb



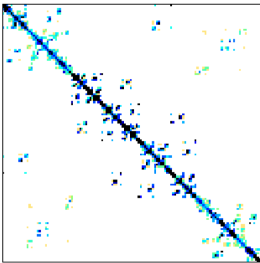
(m) offshore



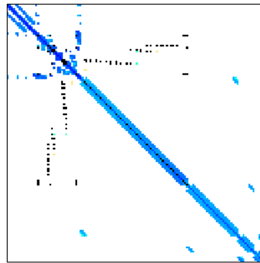
(n) olafu



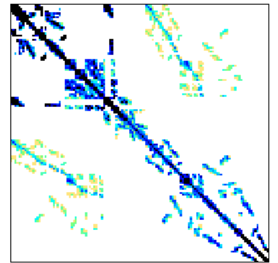
(o) parabolic_fem



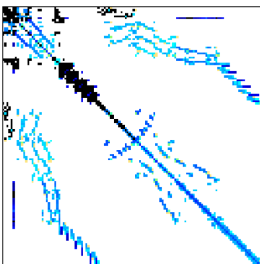
(p) Pres_Poisson



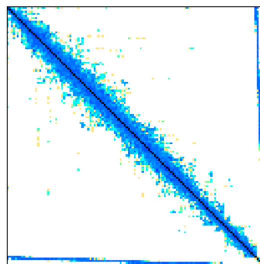
(q) raefsky4



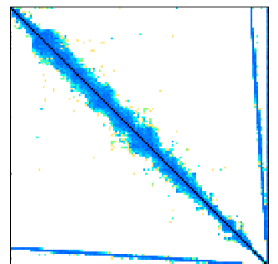
(r) smt



(s) sts4098



(t) thermal1



(u) thermal2

Figure A.1: Matrix Memory Structures (Source: UFlSMC [11])