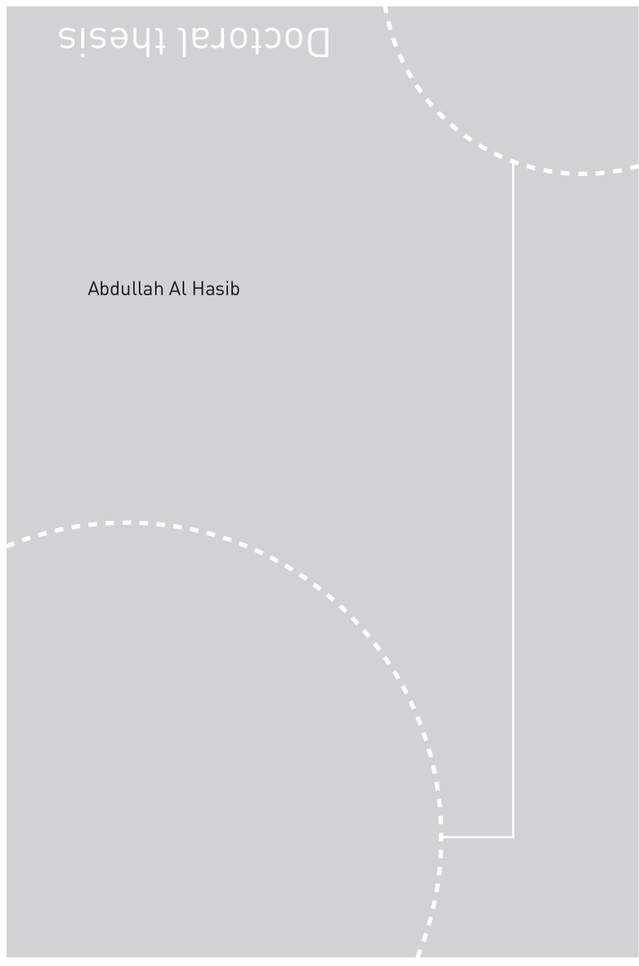
Doctoral theses at NTNU, 2018:143

Abdullah Al Hasib

Energy Efficient Computing on Multi-core Processors

Vectorization and Compression Techniques



ISBN 978-82-326-3076-9 (printed ver.)
ISBN 978-82-326-3077-6 (electronic ver.)
ISSN 1503-8181

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of
Philosophiae Doctor
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Doctoral theses at NTNU, 2018:143

NTNU
Norwegian University of
Science and Technology

Abdullah Al Hasib

Energy Efficient Computing on Multi-core Processors

Vectorization and Compression Techniques

Thesis for the Degree of Philosophiae Doctor

Trondheim, May 2018

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Department of Computer Science

ISBN 978-82-326-3076-9 (printed ver.)
ISBN 978-82-326-3077-6 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2018:143

Printed by NTNU Grafisk senter

This thesis is dedicated to my parents
for their love, endless support and encouragement

Abstract

Over the past few years, energy consumption has become the main limiting factor for computing in general. This has led CPU vendors to aggressively promote parallel computing using multiple cores without significantly increasing the thermal design power of the processor. However, achieving maximum performance and energy efficiency from the available resources on the multi-core and many-core platforms mandates efficient exploitation of the existing and emerging architectural features at the application level.

This thesis presents the study of some of the existing and emerging technologies in order to identify the potential of exploiting these technologies in achieving high performance and energy efficiency for a set of Smart Grid applications on Intel multi-core and many-core platforms.

The first part of this thesis explores the energy efficiency impact of different multi-core programming techniques for a selected set of benchmarks and smart grid applications on Intel SandyBridge and Haswell multi-core processors. These techniques include different parallelism techniques such as thread-level parallelism using OpenMP, task-based parallelism using OmpSs, data parallelism using SIMD (Single Instruction Multiple Data) instruction sets, code optimizations and use of different existing optimized math libraries. In our initial case studies, SIMD vectorization is proven very effective in providing both high performance and energy efficiency.

Though the SIMD vectorization is proven very effective, it can also exert pressure on the available memory bandwidth for some applications like Powel Time-Series Kernel, causing under-utilization of the computing resources and thus energy inefficient executions. In the second part of this research, we investigate the opportunities of improving the performance of SIMD vectorization for memory-bound applications using SIMD data compression, SIMD software prefetching, SIMD shuffling, code-blocking and other code transformation techniques. The key idea is to reduce the

data movement across memory hierarchy by using the idle CPU time. We show that integration of data compression is feasible on the Intel multi-core platforms, as long as we can do it in a reasonable time. We present a comprehensive discussion on the SIMD compression techniques and the code transformations required for achieving efficient SIMD computations for memory/cache bound applications using Powel time series kernel as a demonstrator application.

Finally, we perform feasibility study of SIMD optimization and compression techniques across other application domains using k-means clustering algorithm and full-search motion estimation algorithm. We also extended our experiments on Intel many-core architecture using Intel Xeon Phi co-processor.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfilment of the requirements for the degree of philosophiae doctor (PhD). This doctoral work has been conducted at the Department of Computer Science (IDI), NTNU, Trondheim, Norway. The work has been performed under the supervision of Professor Lasse Natvig.

This PhD in Information Technology has been financed by the Faculty of Information Technology and Electrical Engineering, NTNU, Trondheim.

Acknowledgements

First and foremost, I would like to express profound gratitude to my supervisor Prof. Lasse Natvig for his helpful and invaluable support, encouragement and supervision throughout my Ph.D. studies. I have learned a lot from him and his continuous guidance helped me to complete my PhD studies successfully. I am thankful to my co-supervisors Jørn Amundsen and Magnus Lie Hetland for the assistance in the early phases of the work and for the valuable feedback and encouragement during the mid-term evaluation meetings.

I extend my heartfelt gratitude to my former colleagues and co-authors Prof. P. G. Kjeldsberg, Juan M. Cebrián and Nikita Nikitin for their great guidance, insightful comments and wisdom during the time we had the chance to work together. In addition, my sincere appreciation goes to all my colleagues in the CARD group and in the entire Department of Computer and Information Science, for having created such a stimulating and pleasant working environment.

All of this would not have been possible without the love of my parents. I am as ever, especially indebted to my parents for their unceasing love, trust and support throughout my life. They supported me to follow my dreams with an endless source of morale and encouragement for me.

Finally, I would like to extend my deepest love and gratitude to my wife Sanjeeda her support, encouragement, and patience, which always kept me going, despite hardships. Without her enduring trust, love, and support throughout these years, I would never have completed this work.

Contents

List of Tables

List of Figures

List of Acronyms

ASIC	Application Specific Integrated Circuit
ATLAS	Automatically Tuned Linear Algebra Software
AVX	Advanced Vector Extensions
BLAS	basic linear algebra subprograms
CMOS	Complementary Metal-Oxide Semiconductor
CMP	Chip Multiprocessor
CUDA	Compute Unified Device Architecture
EDP	Energy Delay Product
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HBM	High-Bandwidth Memory
HPC	High Performance Computing
IEC	Intel Energy Checker
MCDRAM	Multi-Channel Dynamic Random Access Memory
MIC	Many Integrated Core
MPI	Message Passing Interface
MSR	Model Specific Register
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PAPI	Performance Application Programming Interface
SIMD	single instruction multiple data
SMT	Simultaneous Multi-threading
SSE	Streaming SIMD Extensions
SVE	Scalable Vector Extension
TBB	Threading Building Blocks
TDP	Thermal Design Power
VPU	Vector Processing Unit

Part I

Overview

Chapter 1

Introduction

1.1 Chip Multiprocessors

The CPU performance in single core architectures is traditionally improved by the use of smaller, faster and more power efficient transistors in conjunction with the innovative techniques like pipelining, branch prediction, out-of-order execution, multilevel cache hierarchy etc. However, as the single core processor hits the power wall [1], the traditional way of improving its performance through frequency scaling and instruction level parallelism becomes too expensive in terms of power and area cost [2]. As a consequence, the enhancement of single core CPU performance is slowing down. To compensate this slowness in performance enhancement and to improve power efficiency, multi-core and many-core architectures are introduced. In these designs, known as *Chip Multiprocessors (CMPs)*, multiple low-power cores are added on a single chip [3]. The underlying concept of CMPs is to improve the throughput of the system by exploiting task-level or thread-level parallelism using multiple cores without significantly increasing the thermal design power (TDP) of the processor [4, 5]. Today, the mainstream processors are equipped with multiple cores to handle computation intensive applications. More complex system architectures are equipped with co-processors or accelerators so that the applications can achieve the maximum efficiency.

Figure 1.1 presents the trend of microprocessors for the last 40 years. Several interesting observations can be made from the figure: First, the clock speed is leveled off in the recent years due to power densities. Second, the transistor counts continues to increase in accordance to Moore's Law [7]

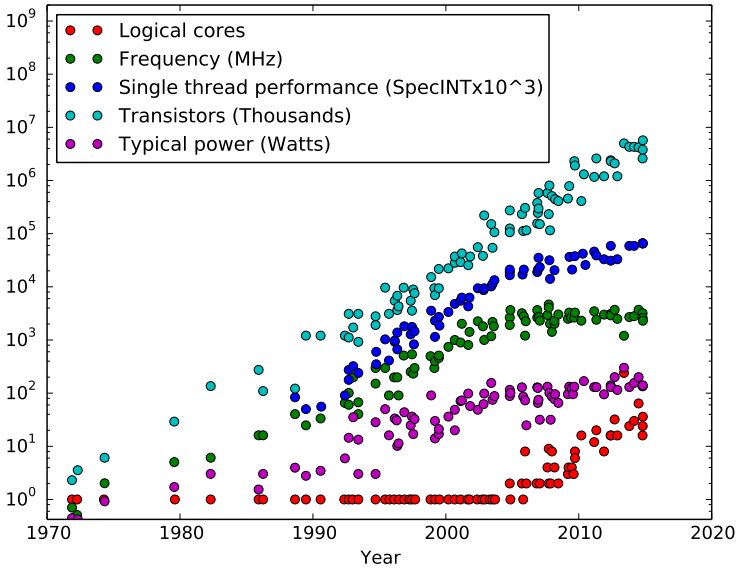


Figure 1.1: Forty years of microprocessor trend data [6]

as the core counts continues to grow for the last decade. Another important observation is that the single-threaded performance has kept increasing slightly. This is attributed due to the use of energy efficient clever power management and dynamic clock frequency adjustments such as Intel Turbo Boost Technology, AMD Turbo Core Technology etc. These technologies enable the processor to run above its base operating frequency if it operates operating below power, current, and temperature specification limits [8].

1.2 Energy Efficient Computation

We have entered an era where CMOS digital computing techniques are reaching physical limits, increasing the importance of energy efficient computations. A large number of research activities have been conducted to develop different energy efficient techniques across different layers in a multi-core system. Low-power circuit design, per-core as well as system wide dynamic voltage and frequency scaling and dynamic power management from operating system level are examples of such techniques [9, 10, 11]. In addition, the unprecedented growth of data, the need for sustainable improvements in computing capacity and energy efficiency in data science, data analytics and scientific computing are becoming greater than

ever. However, it is becoming increasingly difficult to extract more system performance by adding more and more processing cores due to the limitations of *Dennard scaling* [12, 13, 14]. The resulting slowdown in the performance improvement is due to the fact that a significant fraction of the cores has to be switched off (or operated at low frequencies) at any point in time for the power and the thermal limits. This phenomenon is known as *Dark Silicon* [15]. To bridge this dark silicon performance gap, it is absolutely necessary to both improve the energy efficiency of hardware components of the system and to exploit application level energy efficient techniques.

However, for many applications (e.g. commercial applications), it is often undesirable to gain energy efficiency by sacrificing the performance. Therefore, it is also important to realize that energy efficiency does not necessarily mean a process to reduce the energy consumption through compromising the performance [2]. Achieving maximum performance from the available resources on the multi-core platforms mandates the exploitation of all architectural features and their intrinsic parallelism across all granularities [16] such as thread/task-level parallelism and data parallelism. To facilitate thread-level and task-level parallelism, new programming languages, language extensions and libraries are continuously being developed. OpenMP [17], OmpSs [18], Cilk++ [19], Intel Threading Building Blocks (TBB) [20], CUDA [21], OpenCL [18] are some of the popular programming constructs on multi-core and many-core platforms.

Most modern processors also support single instruction multiple data (SIMD) instructions to provide additional throughput and power efficiency through data parallelism [22]. Intel has started supporting 512-bit SIMD computations though Advanced Vector Extensions (AVX-512) [23] and ARM is going to release Scalable Vector Extension (SVE) [24] instruction sets to support up to 2048-bit vectors. Therefore, recent trends clearly show the importance of vectorization in future High Performance Computing (HPC) systems. However, SIMD computations can also turn a CPU bound application into memory bound if the processor runs out of reservation stations or load/store queue entries. Unfortunately, existing software tools and techniques are not often fully able to exploit modern multi-core and many-core architectures [25, 26]. As a consequence, a major research challenge of today is to devise tools and techniques so as to translate the multi- and many-core parallel resources into real application performance.

1.3 Research Context

Smart grids employ information technology to improve the efficiency, reliability and security of the power generation, transmission and distribution processes under the increasing energy demands. In support to the vast potential for development in the Smart Grid sector, NTNU aims at establishing a national center for smart grid research in co-operation with industries, public bodies and other actors.

This dissertation is supported by a PhD fellowship in Smart Grid Research, and was partly performed in cooperation with Powel AS. Powel AS is one of the leading companies in Norway developing solutions to meet the requirements of smart grid technology. One of the smart grid applications studied here is a hydrological time series compute kernel obtained from Powel AS that receives multiple hydrological time series data from database (or reads from file) and produces a summary series.

1.4 Approach and Main Issues

The energy efficiency of computer systems can be improved at various levels and by many approaches. They include low power design and hardware energy saving techniques, system architecture innovation and relevant parallelization techniques, dynamic power management at the OS level, resource scheduling and code optimization using energy aware algorithms and data structures [27] etc. Recent research [22, 28, 16] has revealed the potential of using SIMD execution units to improve the performance and energy-efficiency of compute intensive applications in modern multi-core systems [22, 28, 16]. However, despite the potential of SIMD instructions in developing energy efficient applications, modern compilers still do not have adequate auto-vectorization support for complex codes [29, 30]. Moreover, extracting performance from SIMD and thread-level parallelism is not trivial and a careless implementation can easily obliterate the advantages of modern processors. As a consequence, these powerful processing units (i.e. SIMD units) of modern multi- and many-core systems are often largely underutilized.

It is also important to realize that a significant portion of the overall energy consumption in modern processors is due to data communication across the memory hierarchy [31]. Such consumption is expected to grow even bigger with the unprecedented growth of big data and the emergence of exascale computing. Therefore, it is important to improve the cache effectiveness not only in terms of performance, but also for energy efficiency. SIMD pro-

cessing typically improves the CPU computational power and, if used wisely, can be seen as an opportunity to improve on the application data transfers by compressing/decompressing the data, specially for memory-bound applications.

This dissertation strives to improve energy efficiency of a set of compute intensive as well as memory bound applications by fully exploiting the parallelism potential of SIMD architectures. We primarily focused on designing SIMD friendly data structures and strategies to enhance the utilization of SIMD execution unit as well as to increase the effective cache capacity using compression techniques and on-chip data availability so as to improve the cache hit ratio at the Last Level Cache (LLC). Because of the well known processor-memory performance gap (also called the memory wall), this will significantly reduce the cost of data access and thus enhance the performance and power efficiency.

1.5 Research Questions

Our objective in this thesis is to explore and analyze different multi-core programming methods to improve performance and energy efficiency of some selected smart grid applications. Primarily we will focus on addressing the following research questions:

RQ 1: To what extent multi-core programming, particularly multithreading along with SIMD vectorization, can be suitable for the selected smart grid applications?

RQ 2: To what extent can we improve the energy efficiency of the selected applications by using SIMD compression techniques?

RQ 3: Can we extend our research results to another application domain?

1.6 Thesis Outline

This thesis contains an overview of the research context, research methods and a collection of six papers being produced as a result of this thesis work. The main work and contributions are in the enclosed papers. The overview of the thesis is organized as follows: First, Chapter 2 presents the theoretical background and the relevant research work. Chapter 3 illustrates the methodology used to complete this research work. Chapter 4 contains a brief summary of the included papers. Finally, 5 concludes the thesis with some retrospective remarks and the possibilities of future enhancements.

Chapter 2

Background

This chapter provides more insights about the multi-core and many-core architectures, multi-core programming, and data compression algorithms.

2.1 Multi-core and Many-core Processors

2.1.1 Trends in Parallel Computing Architectures

The demand for faster computers is ever increasing, but the dramatic growth of single processor performance has come to an end due to several technological limitations (such as power wall, memory wall, ILP wall) [3]. To overcome these technological barriers, the leading processor manufactures, from Intel and AMD to Qualcomm and NVIDIA, have shifted their production pipelines from single-core processors to multi-core processors where multiple cores are put-together on a single chip to meet the performance goals without using the maximum operating frequency. As a consequence, multi-core processors result in dramatic increase in the MIPS-per-watt performance compared to that of single-core processors [5, 3].

Today, multi-core processors have become the dominant architecture for the mainstream processors to meet the demand of compute intensive applications. They usually feature up to 24 cores (Table 2.1), but can have beyond 60 cores. Such multi-core processors are often referred to as many-core to express a high core count and are often used for high performance computing (HPC). Often many-core processors have more complex architecture consisting of co-processors, accelerators such as Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) over a PCIe interconnect or even on the same chip.

Table 2.1: Intel CPU architectures and trends

Code name	Processor name	Product name (Intel Xeon)	Frequency (GHz)	Core(s)	Thread(s)	SIMD width	Process technology
Nehalem	Nehalem	W5580	3.20	4	8	128	45 nm
	Westmere	W5680	3.33	6	12	128	32 nm
Sandy Bridge	Sandy Bridge	E3-1280	3.50	4	8	256	32 nm
	Ivy Bridge	E7-4890 v2	2.80	15	30	256	22 nm
Haswell	Haswell	E7-8890 v3	2.50	18	36	256	22 nm
	Broadwell	E7-8890 v4	2.20	24	48	256	14 nm
Skylake	Skylake	E3-1885 v5	3.50	4	8	512	14 nm
Knights Ferry	Knights Ferry	–[1]	1.20	32	128	512	45 nm
Knights Corner	Knights Corner	Phi 7120P	1.24	61	244	512	22 nm
Knights Landing	Knights Landing	Phi 7290F	1.50	72	288	512	14 nm

The recent trend in CPU architectures from Intel is presented in Table 2.1. It is apparent that the evolution of Intel processor architectures has been driven by the increased parallelism for the last decade. As the processor frequency is coming down to reduce power dissipation, the core count is kept increasing to meet the performance requirements using simultaneous multithreading (SMT), also known as hyper-threading which is a commercial trademark for Intel's proprietary implementation of SMT. The emergence of Intel Many Integrated Core architecture (e.g. Knights Corner, Knights Landing) featured with many lightweight cores (tens or even hundreds in near-future) is therefore an attempt to address the problem of improving the energy efficiency without compromising performance.

Also it can be noticed that Intel has given emphasis on achieving higher performance for multimedia and other data-parallel applications through introducing and extending SIMD architectures. Since the introduction of SIMD architectures on general purpose processors, the width of the SIMD registers (SIMD-width) has been increased gradually and now it has become 4-times greater than its initial width. This enhancement can potentially improve the performance of integer and double-precision floating point computations in video encoding, image processing, 3D modeling, scientific simulations, time-series and many other applications by up to 16- and 8-times respectively using Advanced Vector Extensions (AVX-512) instruction set. Moreover, the SIMD units are also augmented with special fused instructions (e.g Fused Multiply-Add, Fused Multiply-Subtract) to improve both performance and energy efficiency.

[1]Product-name is not available, since it is not a commercial product.

2.1.2 Dark Silicon and Heterogeneous Computing System

With the introduction of many- and multi-core architectures, the number of cores on die continues to increase to have enough parallelism for the future applications. However, as Dennard Scaling (i.e. scaling feature sizes and voltages by the same factor) can no longer be maintained with the increasing number of cores, power density (i.e. power consumption per unit area of silicon) continues to rise and is becoming a major limiting factor to the performance of future computing systems. Consequently, these trends might result in next generation chips having more on-chip resources (e.g. processing cores, hardware accelerators, cache blocks and so on) than what can be used simultaneously. In other words, due to the power and thermal limits, a significant fraction of these on-chip resources has to be turned off at any given point in time. This phenomenon is known as *dark silicon* [12, 13, 14]. The challenge is therefore how to make the best use of the abundance of transistors in the dark silicon era [32].

Several techniques have been proposed to mitigate the effect of *dark silicon*. Some of these techniques include near-threshold computing (i.e. operating at a very low voltage to power-on more cores) [33], selective boosting (i.e. satisfy performance requirements by executing boosted cores at the required frequencies for the entire boosting periods, while throttling down the other cores) [34], "Cherry Picking" [35] of cores (i.e. exploiting process variations to select a subset of cores, mapping threads to the selected set of cores and assigning operating frequencies to each core to maximize performance under power budget).

The *dark silicon* issue has also been studied in the context of Heterogeneous computing systems (i.e. systems with many different types of processing cores such as CPUs (Central Processing Units), GPUs (Graphics Processing Units), DSPs (Digital Signal Processors), ASICs (Application Specific Integrated Circuits) etc.). With the integration of specialized processing elements for different types of software applications, the system can dynamically select only a subset of the available processing elements depending on the workload characteristics. Such systems can provide substantial performance and power benefits over general purpose systems [36, 37]. General purpose processors coupled with reconfigurable devices are another promising form of heterogeneous systems that can mitigate the effect of the *dark silicon* phenomenon. Often such heterogeneous components can provide more efficient solutions than full software based solutions on general purpose processing elements. In this context, Microsoft is heading up deploying FPGA-accelerated nodes to improve the performance of computationally ex-

pensive operations in Bing's Indexserve engine [38]. However, the scope of this thesis is limited to Intel multi-core and many-core systems only.

2.2 Multi-core Computing

Multi-core and many-core computing architectures exhibit multiple levels of parallelism through a wide range of architectural features such as multithreading support using multiple cores and data-parallelism (per-core) using SIMD instructions. The effective exploitation of the underlying hardware is therefore crucial for achieving superior performance on current and future processors. For the last few years, researchers have been continuously developing new programming languages, language extensions and libraries to achieve the maximum efficiency by utilizing the modern processors. Here we briefly highlight the key features of some of the well known programming models.

2.2.1 OpenMP

Open Multi-Processing (OpenMP) is the de-facto standard for shared memory programming [39]. It is featured with a set of compiler directives, environment variables and runtime library for explicitly expressing multithreaded parallelism. OpenMP uses fork-join based execution model. It can be classified as mid-level programming model as the implementation details of work-load partitioning, worker management and communication synchronization are kept hidden from the programmer. On the other hand, the programmer has the option to explicitly define scheduling, control the thread affinity, etc. Originally, OpenMP used to provide data-parallelism using these compiler directives. Since the release of OpenMP-3.0, it has started to support task parallelism as well [40].

2.2.2 OmpSs

OmpSs is a task-based programming model and framework focused on supporting heterogeneous multi-core and many-core architectures [18]. This programming model puts an effort to glue the concepts of OpenMP and StarSs [41] together by extending the OpenMP features through incorporating support for asynchronous task-parallelism and heterogeneous architectures. Furthermore, it uses an extended memory model as well as thread-pool execution model instead of fork-join model. Some of the new features of OpenMP 4.0 are inherited from OmpSs. OmpSs is built on top of Mercurium compiler and Nanos++ runtime system [42].

2.2.3 TBB

Threading building blocks (TBB) is another popular C++ runtime library for parallel programming developed by the Intel Corporation [20]. It contains data structures and algorithms used to parallelize an application and to enhance performance on multi-core systems. TBB provides high-level abstraction as it hides the details about the threading mechanisms for performance and scalability. Threading building blocks introduces the concepts of task-stealing, recursive splitting etc. Task-stealing allows TBB to significantly reduce load imbalance as well as improve its performance scalability [43].

2.2.4 MPI

Message Passing Interface (MPI) [44] is the dominant programming model for programming distributed parallel machines. It provides a comprehensive messaging API that can be used to communicate between processes that reside in separate address spaces. Typically, the optimal performance is achieved when each MPI process is mapped on a separate core. For easier coordination among processes, MPI interface provides functionality for communication, synchronization and virtual topology.

2.2.5 Cilk Plus

Cilk Plus is an extension to C/C++ to provide both task and data parallelism [45]. Though Cilk Plus evolved from Cilk, it differs from Cilk in different aspects such as support for loop, C++ language and constructs to solve data race problems. It features simple but powerful ways of specifying parallelism. Cilk Plus provides the `_Cilk_spawn` and `_Cilk_sync` keywords to spawn and synchronize tasks; `_Cilk_for` loop is a parallel replacement for sequential loops in C/C++. Tasks in the Cilk Plus environment are executed within a work-stealing framework. Every worker thread has deque[2] of tasks and the thread treats its deque as a stack, by pushing and popping tasks at the back of it. On the contrary, the thieves steal tasks from the front of the deques [46].

2.2.6 CUDA

CUDA (Compute Unified Device Architecture) is a set of C++ language extensions plus an accompanying runtime API for programming NVIDIA Graphics Processing Units (GPUs) [21]. This programming model is primarily developed for data parallel applications demanding intensive graphics

[2] A double-ended queue

processing. However, it does not provide any viable interface for creating dynamic tasks or handling load balancing issues. Such support has to be entirely coordinated by the programmer. Nevertheless, it includes explicit memory management functions to assist programmers in extracting benefits from the underlying hardware.

CUDA allows programmers to launch large batches of SIMT (Single Instruction Multiple Thread) threads. These threads are hierarchically organized into warps, blocks, and grids, in which the finest thread group (e.g. a warp of 32 threads) runs the same set of instructions in SIMD fashion. Threads in a block are mapped into the same streaming microprocessor (SM) and execute the same kernel in lock-step fashion. Threads in different SMs may run different kernels without any performance penalty though. Finally, the thread-blocks in a grid are executed independently and in an arbitrary order.

2.2.7 OpenCL

OpenCL (Open Computing Language) is an open standard for parallel computing on heterogeneous architecture [47]. One of the main objectives of OpenCL is to increase portability across different platforms ranging from simple embedded micro-controllers to general purpose CPUs and massively-parallel GPGPU hardware pipelines. To this end, OpenCL provides a top level abstraction to the low level hardware as well as consistent memory and execution models. Though OpenCL provides functional code portability, its performance is not always portable across different platforms [48, 49].

2.2.8 Parallel Haskell

Haskell is a pure functional programming language [50]. Haskell offers diverse extensions and libraries for developing parallel or concurrent programs. Some of the well-known extensions for developing parallel applications in Haskell are:

- Glasgow Parallel Haskell: an extension to provide thread-based semi-explicit parallelism on multi-cores [51].
- Accelerate: an embedded domain-specific language for programming the GPU [52].
- HaskellMPI, Glasgow Distributed Haskell, Eden, Cloud Haskell: to develop programs on distributed computing environment [53, 54].

Table 2.2: Summary of a few parallel programming models.

Programming models	Memory model	Parallelism model		Levels of abstraction
		data	task	
OpenMP	shared mem	✓	✓	mid
OmpSs	shared mem	✓	✓	mid
MPI	msg passing	✓	✓	low
Cilk Plus	shared mem	✓	✓	mid
TBB	shared mem	✓	✓	mid
OpenCL	hierar. mem	✓	✓	low
CUDA	hierar. mem	✓	-	low
Parallel Haskell	shared mem/msg passing	✓	✓	high
SIMD	-	✓	-	low

- Meta-par: aims to unify parallel heterogeneous programming using Eden, Cloud Haskell [55].

Table 2.2 briefly summarizes the features of some of the well-known parallel programming models.

2.3 Single Instruction Multiple Data Processing

In the recent years, support for fine grained data parallelism using SIMD instructions is becoming prevalent in virtually every processor in the market. Most of the computers today implement some form of SIMD instruction set. SIMD instructions provide higher performance, better energy efficiency and greater resource utilization [22]. Applications with potential for performance improvement using SIMD instructions are very common. However, despite the potential of SIMD instructions in developing energy efficient applications, modern compilers still do not have adequate auto-vectorization support for complex codes [29, 26]. Therefore, manual vectorization is often required to enhance code efficiency to a large extent.

2.3.1 SIMD Instruction Sets

MMX™, Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are some of the examples of SIMD instruction sets supported by Intel. MMX™, SSE, AVX and AVX512 support 64-bit, 128-bit, 256-bit and 512-bit vector computations respectively. 3DNow! of AMD, NEON of

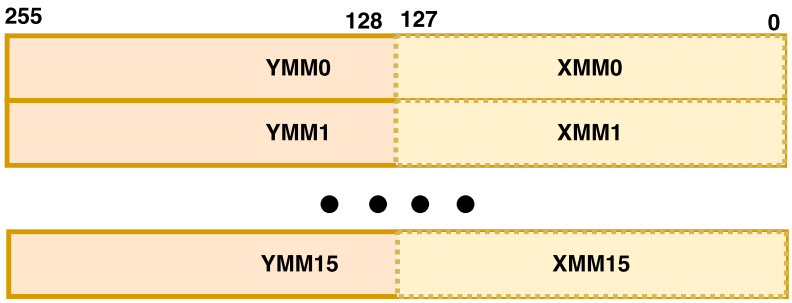


Figure 2.1: AVX register scheme as extension from the SSE (XMM0-XMM15)

ARM, AltiVec of IBM/Motorola are examples of SIMD instruction sets supported by other microprocessor vendors. However, here we briefly present the key features of different SIMD instruction sets for Intel architectures since our experiments have been conducted on Intel processors only.

MMX™

The MMX™ instruction set was introduced in 1997 by Intel on Pentium processors. This instruction set uses 8 64-bit MMX registers and the operations are limited to integer values only. MMX is seldom used in modern processors; SSE and AVX are more commonly used SIMD instruction sets.

Streaming SIMD Extension

Streaming SIMD Extension was introduced with the Pentium III processor in 1999 [56]. This instruction set uses 8 (xmm0 – xmm7) and 16 (xmm0 – xmm15) dedicated 128-bit XMM registers on 32-bit and 64-bit architectures, respectively. SSE supports single-precision floating point operations (but not double precision), and using XMM registers, it is possible to process four floating point number simultaneously. SSE2 introduced in 2000 on Pentium 4 extends SSE by adding support of double precision floating point and integer values (i.e. replaces MMX integer vector instructions) and added over 70 new instructions. SSE4.1, available since the introduction of Intel Penryn [57], consists of 47 instructions for media data manipulation such as Single- and double-precision dot product, streaming load, packed blending, packed integer min/max, sum of absolute differences for 4-byte blocks. SSE4.2 is first available in Nehalem-based Core i7 processors [58] and consists of 7 new instructions for text processing and some application-specific operations such as CRC32 (calculates cyclic redundancy check of a block of data) or POPCNT (counts the number of bits set in a word).

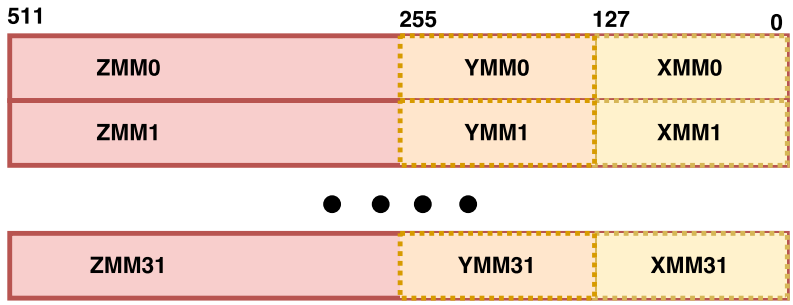


Figure 2.2: AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15)

Advanced Vector eXtension

AVX is the next generation of the SIMD instruction sets supported from Intel Sandy Bridge (i.e. since 2011) processors [59]. It offers instruction sets for YMM registers. There are total 16 YMM registers (ymm0 – ymm15) in the CPUs and the size of each YMM register is twice (i.e. 256-bit) as big as the size of XMM register. The lower 128-bits of these registers are aliased to the respective 128-bit XMM registers. It is possible to perform simultaneous processing of eight single precision floating point numbers or four double precision floating point numbers on these registers. AVX register scheme is illustrated in Figure 2.1. AVX2 extends AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. It also provides enhanced functionality for broadcast/permute operations, vector shift instructions with variable-shift count as well as support for fetching non-contiguous data elements from memory.

Advanced Vector eXtensions-512

The AVX-512 instruction set extends AVX to 512-bit. It was introduced in Intel's Xeon Phi x200 (i.e. Knights Landing in 2016) processor. Advanced Vector eXtension-512 uses 32 512-bit ZMM registers (zmm0 – zmm31) to enable processing of twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE. It is further featured with 8 dedicated mask registers, a new set of blending instructions using mask registers, embedded rounding and broadcast operations, additional gather/scatter support, high speed math instructions (e.g. new fused multiply-add, exponential/reciprocal etc.). Figure 2.2 illustrates AVX-512 register scheme and Table 2.3 presents the evolution of SIMD instruction sets for Intel architecture.

Table 2.3: Evolution of SIMD extensions

Year of Introduction	SIMD Extension	Introduced in	Vector size	Major new features
1997	MMX	Pentium MMX	64-bit	Integer arithmetic operations
1999	SSE	Pentium III Katmal	128-bit	Single precision FP support
2000	SSE2	Pentium 4 Willamette	128-bit	Integer and double precision FP supports
2004	SSE3	Pentium 4 Prescott	128-bit	Complex arithmetic and graphics supports
2007	SSE4.1	Penryn	128-bit	Dot product and conversion
2009	SSE4.2	Nehalem	128-bit	POP-count, CRC
2011	AVX	Sandy Bridge	256-bit	Basic FP and elementary math operations
2013	AVX2	Haswell	256-bit	Extension of SSE(2/4) instructions to 256-bit
2016	AVX512	Knights Landing	512-bit	512-bit extensions to the 256-bit AVX(2)

2.3.2 SIMD Programming Methods

SIMD programming can be mainly done using three methods such as inline assembly [60], intrinsic function [61], and vector class [62].

Inline Assembly

Inline assembly allows a programmer to handle almost every part of the program in order to gain maximum performance benefit from the system. However, writing assembly code is complicated and it demands cautious handling of data transfer between CPUs and memories. Listing 2.1 illustrates AT&T syntax based inline assembly codes [63] to perform summation of two arrays, where the name of the vector registers (e.g. *xmm0*) to be used with the opcodes are explicitly mentioned (e.g. *movups*).

```
1  asm volatile (
2   "movups %1, %% xmm0 \n\t"    //Move values from A[i] to  xmm0
3   "movups %2, %% xmm1 \n\t"    //Move values from B[i] to  xmm1
4   "addps %% xmm0, %% xmm1 \n\t" //Add packed single-precision
       floating-point values from xmm0 to xmm1.
5   "movups %% xmm1, %0" //Move values from xmm1 to  C[i]
6   :
7   "= m" (C[i]) //=m indicates C[i] is an output and it is in
       memory
8   :
9   "m" (A[i]), //m indicates A[i] is an input and it is in memory
10  "m" (B[i]) //m indicates B[i] is an input and it is in memory
11 );
```

Listing 2.1: Inline assembly code to perform summation of two arrays.

SIMD Intrinsic

SIMD intrinsic uses standard C/C++ language and thus coding is easier as compared to the coding with inline assembly. However, it comes with the

cost of not having any guarantee that the code is optimized to the highest level. Listing 2.2 shows SIMD intrinsic codes to sum-up two arrays. As we can see in the example listing, SIMD variables (e.g. xmm_A) are used in the code rather than explicitly using physical register names.

```
1 __m128 xmm_A = _mm_load_ps(A);
2 __m128 xmm_B = _mm_load_ps(B);
3 __m128 xmm_C = _mm_add_ps(xmm_A, xmm_B);
4 _mm_store_ps(C, xmm_C);
```

Listing 2.2: SIMD intrinsic code to perform summation of two arrays.

Vector Class

Vector class is a library that encapsulate SIMD intrinsics for different hardware platforms behind a common interface. Compared to SIMD intrinsic functions, vector class is easier to program with. It also leads to more portable code. However, the downside of this model is that it provides lower performance than the other models. In Listing 2.3, the summation of two arrays is done using vector class. In the listing, the type $F32vec4$ in example is a class that encapsulates the intrinsic type $\_\_m128$ which represents a 128-bit vector register holding 4 floating point numbers of 32 bits each. Moreover, instead of using SIMD instructions or assembly codes, the overloaded operators $+$ are used to add the vectors. Therefore, the code is much easier to understand than the two other models.

```
1 F32vec4 vec_A;
2 F32vec4 vec_B;
3 F32vec4 vec_C;
4 loadu(vec_A, A);
5 loadu(vec_B, B);
6 vec_C = vec_A + vec_B;
7 storeu(C, vec_C);
```

Listing 2.3: Vector class code to perform summation of two arrays.

2.3.3 SIMD Optimization Techniques

The following section briefly covers the different performance optimization techniques using SIMD computations.

Data Packing

Using SIMD (e.g. SSE, AVX, AVX-512) instruction sets and their corresponding vector registers (e.g. 128-bit XMM, 256-bit YMM, 512-bit ZMM), one can pack multiple data into a single XMM/YMM/ZMM register. These data items can be processed simultaneously. Moreover, SIMD instruction sets

also contain special instructions for frequently used- or critical-operations such as matrix transpose, square root, sum of absolute difference etc. to improve the performance even further.

Data Reuse

Vector register space is increasing gradually in the modern processors (e.g. AVX-512 featured microprocessors have 32 ZMM registers representing 2 KB of register space). It is also possible to increase the performance of the SIMD code using data reuse. SIMD intrinsics or inline assembly allows a programmer to have control over this register space. Thus, a programmer can use inline assembly or intrinsics to reuse existing registers effectively so as to reduce unwanted data transfer.

Asynchronous Computation

Data transmission along the memory hierarchy is a slow process. SIMD computation creates additional pressure to the memory hierarchy as it demands simultaneous accesses of more data elements than the scalar computation. Fortunately, this transmission process can be hidden to a certain degree using asynchronous computation, which is a technique to perform computation and data transmission simultaneously [64]. SIMD instructions provide some basic prefetching methods to provide support of asynchronous communication. Prefetching instructions provide hints to CPU for preloading the data into cache before the actual computation begins [65]. It is important to note that these instructions do not force the data to be preloaded into the cache. Thus it is not guaranteed that the data will be preloaded into the cache.

2.4 Data Compression

In modern processors, a significant amount of energy is consumed in the memory hierarchy due to data transmission. For instance, in Xeon Phi Knight's Corner (KNC) co-processor, caches are responsible for around 45% and 12% of the core power distribution for non-compute-intensive and compute-intensive applications, respectively [31]. Therefore, the use of application specific compression methods can increase the effective cache capacity, reduce cache misses and improve the performance and system energy. In this section, we present some of the most widely used loss-less integer compression techniques.

Some of the earliest integer compression techniques are Delta coding [66], Variable-byte (VarByte) coding [67], Rice and Golomb coding and Elias

Gamma coding [68]. *Delta coding* encodes integer numbers by subtracting successive values. The fundamental concept is to code the first value as it is. The remaining values are represented as the differences between successive values. A good compression ratio can be achieved by this approach if the differences between successive values are small. *VarByte* is a simple byte-oriented compression method. It uses a variable number of bytes to represent integer values (e.g. 34 is represented using 8 bits, 144 is represented using 16 bits). This compression technique is fast but it provides low compression ratio. Golomb and Rice coding are bit oriented coding schemes. In *Golomb coding*, an integer (i) is coded by quotient (q) and remainder (r) of division by the divisor (d). In case of *Rice coding*, the divisor used is a power of 2. These coding schemes provide good compression ratio for small integer values but the decompression speed is quite slow. In *Elias Gamma coding*, an integer (i) is encoded by its binary representation preceded by $\lfloor \log_2 i \rfloor$ zeros. This relatively slow technique provides good compression when small integers are more frequent than large integers.

In recent years, several fast compression techniques have been proposed, including the Simple family (S9, S16) [69], Frame Of Reference (FOR) or Patched Frame Of Reference (PFOR) [70]. *Simple9* encoding technique packs as many integers as possible in 32 bits (one word). This encoding technique is fast and provides better compression ratio than *VarByte*. *FOR* and *PFOR coding* compress a block of numbers at a time (e.g. 128 numbers). A fixed number of bits are used to encode each regular number in case of fixed-length encoding. The numbers that cannot be encoded using fixed length bits are considered exceptions and stored using 4 bytes. In *varint-G8IU* [71], a variable number of integers are encoded in 8 bytes. Once encoded, they are grouped together along with a 1-byte descriptor containing the unary representations of the lengths of each encoded integer. 8 data bytes can be used to encode from 2 to 8 integers depending on the size of the encoded integers. The number of integers is encoded by the number of zeros in the descriptor.

More recently, SIMD-based compression techniques have gained popularity. Stepanov et al. presented a taxonomy for variable-length integer encoding formats and presented a byte preserving integer encoding algorithm using SIMD instructions [72]. Schlegel et al. applied SIMD-based decompression algorithms to derive parallel versions of two well-known integer compression techniques: Null Suppression and Elias Gamma encoding [71]. In [73], Ao et al. proposed a linear regression-based parallelized compression technique for lists intersection. In [74], Zhang et al. proposed a parallel com-

pression algorithm that aims to improve the instruction level parallelism by exploiting a 4-way vertical data layout format. Lemire et al. proposed a delta coding-based compression technique, *FastPFOR*, that uses vectorized binary packing over blocks of 128 integers [75]. This scheme stores the exceptions on a per page basis, but selects the base value b on a per block basis. They had shown that their approach is nearly twice as fast as the previously fastest schemes on desktop processors (*varint-G8IU* and *PFOR*). The authors have further extended their work by computing prefix sum using SIMD operations in their proposed algorithm [76].

Chapter 3

Methodology

This chapter discusses the methodology used to conduct this research work. This methodology involves choosing a set of applications to study, exploring different techniques for energy efficient computations and identifying suitable methods for estimating or measuring consumed energy on the target platforms. The rationale behind these selections are explained in this section. Section 3.1 lists the applications and benchmarks that are chosen and explains the reasons behind those selections. Then we discuss the programming languages and techniques that are taken into considerations during design space exploration in Section 3.2. Finally, the evaluation framework used for obtaining and analyzing the results is presented in Section 3.3.

3.1 Test Applications and Benchmarks

A selected set of applications and benchmarks are used in the different experiments presented in this research work. Here, we refer to an "application" as a program designed to perform a group of coordinated functions, tasks, or activities for the benefit of the user such as word processing program. On the other hand, a "benchmark" refers to a program or a set of programs that are commonly used in performance comparison of various subsystems across different chip/system architectures. LINPACK [77] is an example of a benchmark suite used to measure the floating point computing power of a system.

Selection of applications and benchmarks used in this thesis is primarily guided by the following two criteria.

- Research collaboration with industry and academia.

- Relevancy with the experiments as well as ease of integration with the framework.

Application set selection process is predominantly based on the research collaboration with the industry (e.g. Powel AS for Hydrological Timeseries) and internal research group (e.g. Boye Annfelt Høverstad et al. for Data cleansing). On the other hand, the selected benchmarks (FFTW, BlackScholes etc) are widely used, well known representatives of important real-world applications and can be integrated to our evaluation framework.

3.1.1 Benchmarks

1 dimensional FFTW (Fastest Fourier Transform in the West) kernel [78], BlackScholes from the PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark suite [79] and a tiled matrix-matrix multiplication routine [80] are chosen for evaluation.

1D FFTW

The Fastest Fourier Transform in the West (FFTW) is a software benchmark that efficiently implements Discrete Fourier Transform [78]. The implementation takes advantage of the available vector units (e.g. SSE/SSE2/AVX, Altivec) on the underlying platform. Performance of FFT heavily depends on the design of the memory subsystem and how well it is exploited. In general, FFTW delivers reasonably good performance on all hardware due to its self tuning design. In the first step, it creates a plan of execution for the given problem, and then it executes the plan. The plan is chosen by heuristically tailoring execution to the current system (e.g. querying cache sizes), and several different plans are tested to find the fastest candidate.

BlackScholes

BlackScholes is part of the PARSEC benchmark suite [79]. The PARSEC benchmark suite consists of a number of multithreaded applications representing both current and emerging workloads for multi-processing systems. Workloads in the benchmark cover different application domains (e.g. recognition, mining, synthesis and systems application), different parallelism granularity and exhibit different runtime behaviour. BlackScholes is considered as the simplest of all PARSEC workloads [81]. It uses Black-Scholes partial Differential Equation (PDE) to calculate the prices for a portfolio of European-style options. BlackScholes is a compute-bound workload, and its performance is limited by the amount of floating-point calculations a processor can perform.

Matrix Multiplication

Matrix multiplication is widely used in many scientific and engineering applications. Double-precision GEneral Matrix-matrix Multiplication (DGEMM) is an important routine of the LINPACK benchmark [77], which is used to rank the top 500 supercomputers [82]. In this thesis, a tiled matrix multiplication algorithm is used. Matrices are decomposed into tiles of (M$\times$N) elements, and then the DGEMM routine is applied at the tile level. While there exist several basic linear algebra subprograms (BLAS) libraries (such as Intel Math Kernel Library, ATLAS and OpenBLAS) [83], we have used ATLAS (Automatically Tuned Linear Algebra Software) library for the optimized DGEMM routine [84]. Being a portable library, ATLAS automatically tunes itself to the underlying architecture to maximize performance. Furthermore, tiled matrix multiplication also suits very well to the work-sharing constructs of parallel programming models like OpenMP, OmpSs etc.

3.1.2 Smart Grid Applications

Powel Hydrological Time Series Kernel

Powel AS is one of the leading companies in Norway developing solutions to meet the requirements of Smart Grid applications [85]. These solutions contain computer programs and systems for maintaining maps over power lines and pipes, metering of electricity, inflow modeling and forecasting as well as power generation planning. Some of their developed products include [86]:

- Inflow forecasting and power inflow: A program to forecast, simulate and optimize hydropower generation process depending on weather forecasts, snow cover, ground and landscape conditions.
- Hydrometric simulations and Powel Sim/Shop:
  - Powel sim: Short term hydropower planning software to calculate the consequence of a given plan by using inflow forecasts.
  - Powel Shop: A program for short-term optimization of hydropower generation to find how to generate the contracted energy at the lowest possible cost.
- Powel MDMS: Powel meter data management system consists of programs for metering, maintaining and performing computations on different time series data.

In this thesis, we have studied Powel hydrological time series kernel, a part of Powel MDMS which is responsible for generating a summary series from a number of time series data given as input to the system. Each series (s_i) in the input dataset (**ds**) can be formally defined as,

$\mathbf{ds}=\{s_i\}_{i=0}^{n-1}, s_i = (t_i, v_i, q_i)$

where v_i represents the hydrological in-flow measured in m^3/sec at time t_i, and q_i represents the status information (e.g. valid or invalid data) for the measurement at time t_i.

The kernel takes two time series as inputs at a time and performs computations as it passes over the data points at each time instance, generating a new time series as output. Initially, it computes time axis **t** vector of the resultant series from the given inputs. Next, for each t_i in the resultant series, the kernel takes the corresponding v_i from the input series and computes v_i of the resultant series. Any missing value in the input series is estimated using linear interpolation. The corresponding q_i parameter is also updated to indicate that the value is interpolated. Finally, all the aforementioned steps are repeated until the entire dataset (**ds**) is processed.

Powel hydrological time series kernel is implemented in C++. It performs computationally inexpensive $O(n)$ operations and the data access pattern is sequential. However, the kernel needs to process a huge volume of time series data, and consequently it becomes limited by the memory subsystem. Parallelism is handled at a higher level, serving multiple requests simultaneously, further increasing the pressure on the memory hierarchy.

Data Cleansing Application

Data cleansing is a process of detecting and correcting or removing corrupted data in a dataset. Among the 4 different cleansing approaches (statistical methods, clustering, pattern based methods and association rules) [87], we studied a statistical method presented by Chen et al. in [88]. This algorithm is used in a research study to understand the impact of it as a pre-processing step for predicting loads in smart grids.

Data Cleansing model is based on the algorithm proposed by Chen et al. in [88]. The essence of the proposed solution to the detection of corrupted data is to model the intrinsic patterns or structure of load data. The model found can be used to judge the presence of abnormal deviations from the patterns, and thus to identify corruption of data. Assume that *n* data points $(t_i, y_i)_{i=1}^{n}$ of a load curve have been collected. The underlying data gener-

ation process is modeled as a continuous function.

$$y_i = m(t_i) + \epsilon_i \tag{3.1}$$

where y_i is the data value at time t_i, $m(t)$ is the underlying function and ϵ_i is the error term. It is assumed that the error term ϵ_i is normally and independently distributed with the mean of zero and constant variance σ^2. The main task is to find an appropriate estimate of the function $m(t)$, namely $(\hat{m})(t)$, using the collected load data as a sample of observations. Then the point-wise confidence interval can be built based on the estimate of $m(t)$. A data point is judged as corrupted if it locates outside the confidence interval. Corrupted data at t time will be replaced by the estimated value $(\hat{m})(t)$.

A smoothing parameter is used to control the curve smoothness. A smoother curve $m(t)$ tends to model global patterns since it is less sensitive to local deviations, whereas a rougher curve $m(t)$ is more capable of modeling local patterns. Different settings of the smoothing parameter can be chosen to model global or local patterns.

The basic idea of nonparametric smoothing is the local averaging procedure. Specifically, the curve can be estimated by

$$\hat{m}(t_i) = \frac{1}{n} \sum_{i=1}^{n} W_i(t) y_i \tag{3.2}$$

where $\{W_i(t)\}_{n=1}^{n}$ denotes a sequence of weights which depend on the whole vector $\{T_i\}_{n=1}^{n}$. The idea is to approximate the function $m(t)$ by taking a weighted sum or linear combination of a sufficiently large number K of basis functions $\varphi_k(t)$

$$m(t_i) = \sum_{k=1}^{K} C_k \varphi_k(t) \tag{3.3}$$

where **C**=(c_1, c_2, ..., c_K) is the coefficient vector. To estimate coefficients **C** from the observations $(t_i, y_i)_{i=1}^{n}$, we define an *n* by *K* matrix

$$\Phi = \begin{vmatrix} \varphi_1(t_1) & \varphi_2(t_1) & \dots & \varphi_K(t_1) \\ \varphi_1(t_2) & \varphi_2(t_2) & \dots & \varphi_K(t_2) \\ \vdots & \vdots & \vdots & \vdots \\ \varphi_1(t_n) & \varphi_2(t_n) & \dots & \varphi_K(t_n) \end{vmatrix} \tag{3.4}$$

A simple smoother could be obtained if the coefficients **C** are determined by minimizing the Sum of Squared Error (SSE) as

$$SSE = \sum_{j=1}^{n} \left| y_j - \sum_{k=1}^{K} C_k \varphi_k(t) \right|^2 \tag{3.5}$$

$$m(t_i) = \sum_{k=1}^{K} C_k \varphi_k(t) = (\vec{y} - \Phi^T \vec{c})^T (\vec{y} - \Phi^T \vec{c}) \tag{3.6}$$

Now, the coefficients can be estimated by minimizing the penalized sum of squared errors (PENSSE). PENSSE is given by

$$PENSSE_\lambda = (\vec{y} - \Phi^T \vec{c})^T (\vec{y} - \Phi^T \vec{c}) + \lambda \vec{c}^T R \vec{c} \tag{3.7}$$

Then the coefficients $\vec{c}$ can be estimated by setting the derivative of PENSSE with respect to $\vec{c}$ to be zero

$$\hat{\vec{c}} = (\Phi^T \Phi + \lambda R)^{-1} \Phi^T \vec{y} \tag{3.8}$$

where λ is the smoothing parameter and $R = \int D^2\vec{\Phi}(t) D^2\vec{\Phi}(t)^T dt$.

Finally, the fitted value vector $\hat{\vec{y}}$ is computed by

$$\hat{\vec{y}} = \Phi\hat{\vec{c}} = \Phi(\Phi^T \Phi + \lambda R)^{-1} \Phi^T \vec{y} \tag{3.9}$$

3.1.3 Applications from Different Domain

Apart from the aforementioned applications, we have also considered data mining (k-means) and multimedia (motion estimation) applications in our evaluation process.

K-means Algorithm

K-means is one of the most widely used clustering methods. Among the different variants to this algorithm, we set our baseline as Lloyd's algorithm [89]. The advantage of this algorithm is its simplicity: starting with a set of randomly chosen initial centers, the kernel repeatedly assigns each input point to its nearest center, and then recomputes the centers given the point assignment. However, the algorithm is sensitive to its initialization process in obtaining a good solution. In [90], a randomized seeding technique is proposed to improve the accuracy and speed of k-means algorithm.

More recent works involve optimizing k-means computations using distance bounds and triangular inequality [91]. In [92], a fine-grained SIMD based approach is proposed which computes n distances from the n data points to the same centroid in one loop. In [93], Hadian and Shahrivari have used a KD-tree (k-dimensional tree) based structure where each node for the KD-tree is represented by a bounding box specifying the minimal axis-parallel hyper-rectangle containing all associated points. Consequently, the search for nearest centroid is accelerated. In [94, 95], the authors proposed an algorithm that performs the distance calculations in parallel on the GPU while sequentially updating the cluster centroids on the CPU based on the results from the GPU calculations. In these optimization methods, parallelism is done at the task level, where the data is divided into smaller chunks and each chunk is processed in sub-tasks. All of these tasks execute the same logic as in the Lloyd's algorithm.

Full Search Motion Estimation

Motion Estimation (ME) is a core part of different video compression algorithms. Block-based ME algorithms involve finding the candidate block within a specified search area in a reference frame that is most similar to the current block in the current frame. A "full-search motion estimation" algorithm performs an exhaustive search over the entire search region to find the optimal solution. This process is computationally intensive and costs about 80% of the video frame encoding time [96]. Therefore, it has been chosen as one of our test applications in the thesis.

3.2 Design Space Exploration

An important goal of this thesis is to improve performance and energy efficiency of a selected group of applications on multi-core and many-core platforms. To this end, we explored a number of state-of-the-art multi-core programming techniques (e.g. OpenMP, SIMD programming, optimized lib-

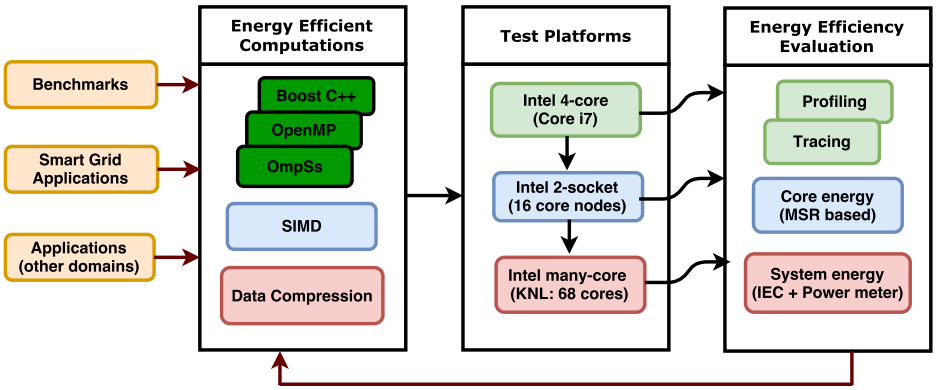


Figure 3.1: Research methodology to conduct this research

raries such as ATLAS, MKL) and followed some of the promising optimization strategies (e.g. loop transformation, code reorganization, blocking, lookup table usage) to port the selected set of applications and benchmarks onto modern multi-core desktop processing systems. While doing so, we focused on providing interesting insights on the effects of using these programming tools and techniques on applications' performance and energy efficiency. Essentially, we focused on addressing the research question 1 (RQ-1) at this point.

The experimental evaluations of the studied techniques on different multi-core systems provided satisfactory performance and energy efficiency improvements for regular data processing (e.g. data cleansing application with regular loop structures). Particularly, SIMD vectorization appeared to be a promising means of energy efficient software development. Unfortunately, auto-vectorization was not fruitful for some of the selected applications, such as applications with more complex loops (e.g. Powel time series kernel). In such situations where codes are not prepared for SIMD, we resorted to explicit vectorization which often involved algorithmic elements to meet SIMD requirements, choosing appropriate data layouts and finally writing down the codes using SIMD intrinsics.

The results of explicit vectorization were mixed: for the CPU-bound applications such as motion estimation kernel, the impact of explicit vectorization on performance and energy efficiency was quite impressive. Where as for the memory-bound applications such as Powel time series kernel, the overall performance was lower than the expected level. It turned out that the eval-

uated platform was unable to achieve memory bandwidth necessary for the Powel time series kernel to perform simultaneous memory accesses for the vector elements, thus creating a new bottleneck in the code. In an attempt to improve performance and energy efficiency of memory-bound applications, we explored different SIMD compression techniques so as to trade idle time for computation time by harnessing the power of the (usually) underutilized vector processing units. We have also explored several non-trivial low-level optimization strategies such as controlling thread-affinity and fixing threads into the cores to reduce unnecessary data movement across memory hierarchy, improving cache line utilization ensuring that the adjacent data is actually used in the hot loops (i.e. where an application spends most of its time), hiding data access latency by using SIMD software-prefetching if data access patterns are predictable etc. Thus, we sought the answer of research question 2 (RQ-2) by exploring some non-trivial strategies including data compression techniques to improve energy efficiency of those applications.

Finally, some of these energy efficient approaches are also deployed to k-means algorithm as a part of addressing the research question 3 (RQ-3). Figure 3.1 illustrates the research methodology used in this thesis. We further extended our experiments on Intel many-core platform (Knights Landing Co-processor) [97].

3.3 Evaluation Framework

In this section, we describe the hardware platforms on which the research is conducted. In addition, we also describe the process we follow to track micro-architectural activities as well as the tools and experimental framework used in our research for evaluating performance and energy efficiency.

3.3.1 Test Platforms

Intel® Core™ i7-2600 and Intel® Core™ i7-4700K are chosen for multi-core processors, and Intel® Xeon Phi 7250 is chosen as a many-core processor. This selection process is primarily guided by our research group.

Intel Multi-core Processors

For multi-core CPUs, Intel® Core™ i7-2600 (Sandy Bridge) and i7-4700K (Haswell) processors are used in our evaluation framework. Both processors consist of four physical cores and supports Hyper-Threading (HT) that allows them to simultaneously process up to 8 threads i.e., 2 threads per core. The memory hierarchy consists of a 32 KB Level-1 cache, a 256

Table 3.1: Hardware specifications of the test platforms

Processor	**Intel® Core™ i7-2600**	**Intel® Core™ i7-4700K**	**Intel® Xeon Phi 7250**
Architecture	SandyBridge	Haswell	Knights Landing
Clock Speed	1.6 – 3.4 GHz	0.8 – 3.5 GHz	1.4 Ghz
# of Cores	4 cores / 8 threads		68 cores / 272 threads
L1 Cache	32 KB data + 32 KB instruction, private, 8-way associativity		
L2 Cache	256 KB, private, 8-way associativity		1 MB, 16-way per 2 cores[1]
L3 Cache	8 MB, shared, 16-way associativity		16 GB, shared HBM-MCDRAM[2]

KB Level-2 cache and a 8192 KB Level-3 cache. Level-1 and Level-2 caches are private to each core while the Level-3 cache is shared among the cores. The base clock speeds of the processors are 3.4 GHz and 3.5 GHz for Sandy Bridge and Haswell processors, respectively. When Turbo Boost is enabled, the clock speeds can be as high as 3.9 GHz [8]. However, we disabled dynamic frequency scaling (speed step and turbo boost) on both systems to get more stable experimental results. Table 3.1 illustrates the hardware specifications of the test platforms.

Intel Xeon Phi Co-processor

Our evaluation platform also consists of a Xeon Phi 7250 processor (Knights Landing) featuring the second generation of Intel's Many Integrated Core Architecture (MIC) [97]. This product offers 68 cores running at 1.40 GHz base clock speed and can have up to four threads per core with the potential peak performance close to 6 Tflops for single precision floating point. These cores are based on the *Silvermont Atom* architecture, which is Intel's first low power core built at 22nm process technology. Cores are out of order and tiled in pairs, where each pair of cores shares 1-Mbyte level-2 (L2) cache. Each core contains two Vector Processing Units (VPUs), that work with vector registers up to 512-bit wide. The VPUs are compatible with SSE, AVX/AVX2 and AVX512. The memory architecture of KNL is designed to support its large computational capability. KNL is equipped with High-Bandwidth Memory (HBM) based on the Multi-Channel Dynamic Random Access Memory (MCDRAM). MCDRAM is capable of delivering $\approx$4x performance ($\geq$ 400 GB/s) than DDR4 memory on the same platform ($\geq$ 90 GB/s).

[1] Two cores share a 16-way associative, 1-Mbyte unified L2 cache

[2] Multi-Channel High Bandwidth ($\approx$4x higher bandwidth than DDR4) DRAM

Table 3.2: Used PAPI event-set to monitor cache and memory related events

Counters	Counting
PAPI_L2_TCM	L2 total cache misses
PAPI_L2_TCA	L2 total cache accesses
PAPI_L3_TCM	L3 cache misses
PAPI_TOT_INS	Instructions completed
PAPI_TOT_CYC	Total cycles
RESOURCE_STALLS:ANY	Cycles stalled due to any resource related issue
RESOURCE_STALLS:SB	Cycles stalled due to lack of Store Buffers
RESOURCE_STALLS:RS	Cycles stalled due to no eligible Reservation Station entry
RESOURCE_STALLS:ROB	Cycles stalled as the Re-Order Buffer is full
UOPS_RETIRED:STALL_CYCLES	Cycles without micro-operations retired
PERF_COUNT_HW_CACHE_L1D:MISS	L1 data cache misses
PERF_COUNT_HW_CACHE_L1D:ACCESS	L1 data cache accesses
PERF_COUNT_HW_CACHE_L1D:WRITE	L1 data cache writes

3.3.2 Profiling and Tracing

Profiling (i.e. summary statistics of performance metrics done by sampling) of different micro-architectural activities is performed using Performance Application Programming Interface (PAPI) tool on our test platforms [98]. PAPI relies on the hardware performance counters that are available on most processors and provides a connection between software performance and processor events. Table 3.2 presents a set of PAPI counters that are used in this thesis to track cache and memory related events. The first five events in the list are preset events that are accessible through PAPI interface. Rest of the events in the list are platform dependent events and termed as native events. Generally, each experiment is repeated for multiple times, outliers are discarded and then mean value is computed to report the results.

DineroIV trace-driven cache simulator is used to measure compulsory, capacity and conflict cache misses [99]. We use the Lackey tool of Valgrind [100] to produce memory traces (i.e. chronological records of events while an application is running) to be analyzed by the DineroIV cache simulator. However, we had to modify the Lackey tool to generate the trace format

supported by DineroIV simulator.

3.3.3 Core Energy Estimation

To estimate on-chip energy consumption, an energy collection framework is developed as part of this thesis work [101]. Initially, the framework was developed to read the model specific registers (MSRs) for providing energy measurements on a single socket Sandy Bridge system. Afterwards, the framework has been extended to support dual socket Sandy Bridge as well as Haswell systems. The developed framework is used to provide both core- and package-energy readings from the supporting systems. In this context, the 'core' refers to the components of a processor that are involved in executing instructions such as arithmetic logic unit, floating point unit, L1 and L2 caches etc. On the other hand, the 'package' refers to L3 cache, quick-path interconnect controller, on-die memory controller, and other bus controllers such as PCI Express including the core components [102]. The framework reads a list of registers presented in Table 3.3 at a fixed core frequency, and processes the raw data to compute energy efficiency. The details of the energy efficiency metric is described in section 3.3.5.

Table 3.3: Model Specific Registers for energy measurements

Register	Measured Energy
MSR_PP0_ENERGY_STATUS	Core energy consumption
MSR_PKG_ENERGY_STATUS	Package energy consumption

For KNL, the whole package energy (including core power and DRAM controller traffic) is measured since the core energy counter is not available in our pre-production system. These measurements can be done either by the *RAPL* interface (root-level) or the *powercap* interface (user-level).

3.3.4 System Energy Estimation

To measure system energy consumption, we use an external power meter (Yokogawa WT210) which is connected to a power outlet and an energy server (part of the Intel Energy Checker (IEC) SDK) [103] system to log the consumed energy at a certain interval. Measured readings are retrieved by the Intel Energy Server process running on the energy server (ESRV) system and written to a text file. Finally, the raw data written in the text file is further processed to transfer the raw data into desired evaluation metrics.

3.3.5 Evaluation of Energy Efficiency

We use execution time (in micro-seconds) as the metric for performance evaluation. Energy efficiency is measured in terms of Energy Delay Product (EDP: *Js*) [101]. Generally, the lower the EDP, the better the energy efficiency. Speedup and Relative EDP at a certain frequency are computed with respect to the execution time and EDP of the *Baseline* kernel at the corresponding frequency.

Chapter 4

Research Summary

This thesis is a collection of papers that I have authored or coauthored during my PhD research work. This chapter provides an overview of these papers. First, Section 4.1 briefly sketches the research themes and the publications. Section 4.2 presents more detailed outlines of the included papers in the perspective of what they intend to achieve, and how do they relate to the topic of this thesis. Finally, Section 4.3 lists the papers that were not included in this thesis.

4.1 Research Process

4.1.1 Formalities

The research work described in this thesis was part of a four-year PhD research fellowship programme conducted at the Department of Computer and Information Science, Faculty of Information Technology, Mathematics and Electrical Engineering, NTNU. 25% of this PhD period was dedicated to teaching duty.

4.1.2 Publications and Research Themes

The main contributions of this thesis are published into different international peer reviewed conference proceedings and journals. These papers are grouped into 4 different categories which are presented in Table 4.1. The research focus of each group is discussed in this section.

Table 4.1: Paper categories

Category	Name	Papers	
		Total	Included
A	Programming model exploration	3	3
B	SIMD-vectorization of a smart grid application	1	1
C	SIMD-vectorization of applications from different application domains	2	2
D	Demand Response optimization for smart grids	2	0

Category A: Programming Model Exploration

The first research topic is represented in Category A and relates to the first research question (RQ1). The papers in this category are originated from the initial investigation on state of the art parallel programming models, libraries and extensions. The papers included in Category A are summarized and referenced in Table 4.2.

Table 4.2: Paper category A

ID	Title	Ref.
A.1	Case Studies of Multi-core Energy Efficiency in Task Based Programs	[101]
A.2	Performance and Power Efficiency Analysis of Data Reuse Transformation Methodology on Multi-core Processor	[104]
A.3	Performance Optimization and Evaluation of a Data Cleansing Algorithm on Multi-core Processors	[105]

In paper A.1, we performed energy efficiency case studies of a task based programming model (i.e. OmpSs) on some selected benchmark applications. The study provided insights on the impact of task based programming model on applications' energy efficiency. We further explored the effectiveness of using SSE and AVX vectorization on the selected benchmarks. In paper A.2, we integrated different data reuse transformation strategies with OpenMP programming model and performed an energy efficiency analysis of different transformation strategies. In the next paper, we made a comparative study of the state of the art BLAS libraries for a data cleansing algorithm. We made several implementations of the data cleansing algorithm

using different BLAS libraries and analyzed the impact of those BLAS libraries on its performance and energy efficiency.

In our initial investigations on different multi-core programming models and techniques, the SIMD vectorization appeared to be a promising means of providing energy efficient solutions for certain applications (in our case, BlackScholes and FFTW benchmarks). As a consequence, we have decided to use SIMD vectorization in order to improve the energy efficiency of the selected applications.

Category B: SIMD-vectorization of Smart Grid Application

The paper in category B focused on addressing the second research question (RQ2). To this end, our initial step was to profile the target application (i.e. time series compute kernel) in order to identify the critical parts of it. In the next step, we vectorized the identified critical parts of the application in order to improve its performance and energy efficiency. However, the vectorization did not pay-off as we expected. An in-depth investigation of the application properties using micro-architectural activities (PAPI counters) revealed that the SIMD computations increased the pressure on the memory subsystem, turning the application more and more memory/cache bandwidth limited (e.g. CPU cycles stalls accounted for around 80% of the total CPU cycles).

Table 4.3: Paper category B

ID	Title	Ref.
B.1	V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series	[106]

To address this problem, we presented a vectorized differential compression method (V-PFORDelta) for cache/memory bound compute kernels (Table 4.3, Paper B.1). Our strategy is to increase the cache block utilization and to reduce the total number of off-chip memory accesses by using a lightweight real-time SIMD-based compression/decompression method. V-PFORDelta is based on a hybrid data structure for the compressed data (AoS[1] + SoA[2], aka AoSoA[3]), regarded best practice in SIMD programming [107]. Our proposal is also enhanced with SIMD prefetching to increase the data locality of the application.

[1] Array of Structures.
[2] Structure of Arrays.
[3] Array of Struct of Arrays or Tiled Array of Structs.

Category C: SIMD-vectorization of Applications from Different Application Domains

Once we improved the performance and energy efficiency using SIMD compression on a time series compute kernel, we were interested in whether our findings can be employed into applications from other domains as well. Thereby, we aimed at addressing the third research question (RQ3). Table 4.4 lists papers that are relevant to RQ3.

Table 4.4: Paper category C

ID	Title	Ref.
C.1	A Vectorized k-means Algorithm for Compressed Datasets: Design and Experimental Analysis	[108]
C.2	Energy Efficiency Effects of Vectorization in Data Reuse Transformations for Many-core Processors - A Case Study	[109]

Paper C.1 particularly focused on exploring the effectiveness of SIMD compression on different application domains. To this end, we chose k-means algorithm to study which is one of the most influential data mining algorithms. We made an efficient implementation of k-means algorithm by integrating a lightweight SIMD compression method into k-means to reduce the total required number of memory accesses. We further enhanced its performance by optimizing its loop traversal scheme and introducing an in-register implementation of the most time-consuming function to optimize data locality and conserve memory bandwidth. Paper C.2 was an extension of our previous work done in Paper A.2. We extended our research work by analyzing the effects of parallelism at different granularities by combining vectorization with multithreading. We consider both multi-core and many-core system architectures in our study. Our experiments provided clear indications that data reuse methodology in combination with a parallel programming model can significantly save energy as well as improve performance of this type of applications running on multi-core processors.

Category D: Demand Response Optimization for Smart Grids

Papers in category D, as listed in Table 4.5, were produced by the author, but not included in this PhD thesis. These papers present extended models of demand response (DR), which is an indispensable part of Smart Grid technology. In paper D.I, we argued for the need to consider bidirectional energy trading in DR and presented an efficient linear model for appliance scheduling in a residential building with a hybrid power supply system and an energy storage unit. In paper D.2, we proposed an ILP-based scheduling

algorithm for the presented model that minimizes the cost of energy consumption while maximizing the comfort satisfaction in accordance with the user-willingness to pay for comfort.

Table 4.5: Paper category D

ID	Title	Ref.
D.1	Load Scheduling in Smart Buildings with Bidirectional Energy Trading	[110]
D.2	Cost-Comfort Balancing in a Smart Residential Building with Bidirectional Energy Trading	[111]

4.2 Research Results

In this section, we present an overview of the papers included in this thesis. The included papers are discussed in sections 4.2.1 through 4.2.6. These sections contain the abstract of the paper and a description of the contributions of the different co-authors. Most of the sections also contain a discussion on how the paper is viewed in retrospective.

4.2.1 A.1

Case Studies of Multi-core Energy Efficiency in Task Based Programs
H. Lien, L. Natvig, A. Al Hasib and J. C. Meyer
International Conference on ICT as Key Technology against Global Warming
2012

Abstract

In this paper, we present three performance and energy case studies of benchmark applications in the OmpSs environment for task based programming. Different parallel and vectorized implementations are evaluated on an Intel® Core™ i7-2600 quad-core processor. Using FLOPS/W derived from chip MSR registers, we find AVX code to be clearly most energy efficient in general. The peak on-chip GFLOPS/W rates are: Black-Scholes (BS) 0.89, FFTW 1.38 and Matrix Multiply (MM) 1.97. Experiments cover variable degrees of thread parallelism and different OmpSs task pool scheduling policies. We find that maximum energy efficiency for small and medium sized problems is obtained by limiting the number of parallel threads. Com-

parison of AVX variants with non-vectorized code shows $\approx$ 6 – 7x (BS) and $\approx$ 3 – 5x (FFTW) improvements in on-chip energy efficiency, depending on the problem size and degree of multithreading.

Retrospective View

This paper presented energy efficiency results of task-based parallelism and vectorization for three benchmark applications. At the time when the paper was written, the energy efficiency analysis of task-based programming model was largely an unexplored area. However, the implications of this study might have been even wider and more comprehensive if we could compare OmpSs results with other state-of-the-art programming models (e.g. OpenMP) or estimate the overhead of OmpSs.

Roles of the Authors

Lien and Natvig came up with the initial idea and planned which experiments to be carried out. Lien also conducted most of the experiments and generated corresponding plots as part of his master's thesis, while I contributed to the paper by developing the energy estimation tool we used for measuring core energy as well as by implementing and conducting the matrix multiplication experiment.

Lien prepared the first draft of the paper while I contributed to the paper by describing the energy measurement method. Natvig and Meyer gave advice on the benchmark applications and provided valuable comments which improved the overall quality of the paper.

4.2.2 A.2

Performance and Power Efficiency Analysis of Data Reuse Transformation Methodology on Multi-core Processor
A. Al Hasib, P. G. Kjeldsberg and L. Natvig
Euro-Par 2012: Parallel Processing Workshops
2012

Abstract

Memory latency and energy efficiency are two key constraints to high performance computing systems. Data reuse transformations aim at reducing memory latency by exploiting temporal locality in data accesses. Simultaneously, modern multi-core processors provide the opportunity of improving performance with reduced energy dissipation through parallelization. In

this paper, we investigate to what extent data reuse transformations in combination with a parallel programming model in a multi-core processor can meet the challenges of memory latency and energy efficiency constraints. As a test case, a "full-search motion estimation" kernel is run on the Intel® Core™ i7-2600 processor. Energy Delay Product (EDP) is used as a metric to compare energy efficiency. Achieved results show that performance and energy efficiency can be improved by a factor of more than 6 and 15, respectively, by exploiting a data reuse transformation methodology and parallel programming model in a multi-core system.

Retrospective View

This paper was prepared to fulfill the requirement of a PhD course work, where we investigated performance and energy efficiency effects of applying data-reuse transformations on a multi-core processor running a motion estimation algorithm. The presentation in this paper could have been better. In particular, the background information could be improved to allow the reader to understand the context and significance of the data transfer and storage exploration methodology. Moreover, from the perspective of this thesis work, studying the effect of vectorization and performing scalability analysis would have been very interesting. To this end, we continued this work, which resulted in another research publication (Paper C.2).

Roles of the Authors

The initial idea and preliminary investigations were carried out by me. The idea was then further refined through discussions with Kjeldsberg and Natvig. Then I implemented the refined idea in our test framework, carried out the planned experiments and analyzed the results.

I wrote the first draft of the paper. Kjeldsberg and Natvig contributed with improvements of the paper.

4.2.3 A.3

Performance Optimization and Evaluation of a Data Cleansing Algorithm on Multi-core Processors
A. Al Hasib and L. Natvig
Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems
2013

Abstract

Due to the unceasing rising consciousness of energy and environment as well as the requirement of high quality and reliable power supply for consumers, smart grid has become a common aim of power electric development throughout the world. Reliable and accurate power prediction schemes are the basis of the smart grid technology. These prediction schemes are generally based on complex formulations that are computationally expensive but often need to be solved in shorter time scales. In this paper, we have studied the data cleansing step of a smart grid application, and then applied several optimization techniques to accelerate its performance. Finally we have evaluated the performance and energy efficiency of the implemented algorithm on two multi-core platforms.

Retrospective View

This article featured preliminary survey of a set of BLAS libraries on a data cleansing algorithm. The scope of paper was limited to making an efficient implementation of the data cleansing algorithm and analyzing its performance and energy efficiency on Intel multi-core platforms. Data cleansing is often used as a preprocessing step of load prediction method and it is in general compute intensive process. Therefore, the contribution of the paper could be improved by studying the impact of the performance of this preprocessing step on short- and long-term load prediction process in the smart grids. Additionally, it would have been interesting to perform a full system energy efficiency evaluation rather than limiting it only to the core energy evaluation. Moreover, this study could have been further improved by performing the same experiments on many-core platforms.

Roles of the Authors

The initial idea and preliminary investigations were carried out by me. The idea was then further refined through discussion with Natvig. Afterwards, I implemented the refined idea in our test framework, carried out the planned experiments and analyzed the results.

I wrote the first draft of the paper. Natvig contributed with supervision and comments to the presentation, organization and language.

4.2.4 B.1

V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series
A. Al Hasib, J. M. Cebrián and L. Natvig
International Conference on High Performance Computing
2015

Abstract

Chip multiprocessors (CMPs) and heterogeneous architectures have become predominant in all market segments, from embedded to high performance computing. These architectures exacerbate on-chip data requirements, creating additional pressure on the memory subsystem. Consequently, efficient utilization of on-chip memory space becomes critical for data intensive applications. A promising means of addressing this challenge is to use an effective compression method to reduce the data transmitted along the memory hierarchy.

In this paper we present V-PFORDelta, a real-time vectorized integer differential compression method for memory bound applications. We evaluate the effectiveness of our SIMD (Single Instruction Multiple Data stream) based compression method on an industrial hydrological time series data processing kernel. We analyzed both Streaming SIMD Extensions (SSE) and Advanced Vector Extensions 2 (AVX2) versions of the compression method. Results show that the performance and energy efficiency can be improved up to a factor of 3.1 and 8.2, respectively. The proposed method not only outperforms the uncompressed SIMD implementations of the hydrological kernel, but also reduces the data storage requirements by a factor of 1.56x to 3.38x, depending on the analyzed dataset.

Retrospective View

In this paper, we demonstrated that the approach of using a SIMD-based integer compression algorithm in a memory bound application is feasible, as long as decompression can be performed in a reasonable time. However, there is still some room for improvement with respect to the evaluation of the proposed method. For example, validating the feasibility of the proposed approach using a diverse set of real-world datasets would have strengthened our claim. Another aspect that was not covered is the evaluation of our approach on many-core platform. Nevertheless, the continuation of this work for a different application domain lead to another

research publication (Paper C.1).

Roles of the Authors

I carried out the preliminary investigation and came up with the initial idea. The initial idea was then further refined through discussions with Cebrián and Natvig. Next I planned which experiments to be carried out, conducted the planned experiments and analyzed the obtained results. Cebrián worked as an advisor and provided necessary guidelines during the experiments.

First draft of the paper was prepared by me. Cebrián and Natvig provided valuable feedback on the draft to improve the overall quality of the paper.

4.2.5 C.1

A Vectorized K-means Algorithm for Compressed Datasets: Design and Experimental Analysis
A. Al Hasib, J. M. Cebrián and L. Natvig
Journal of Supercomputing
2018

Abstract

Clustering algorithms (i.e., gaussian mixture models, k-means, etc.) tackle the problem of grouping a set of elements in such a way that elements from the same group (or cluster) have more similar properties to each other than to those elements in other clusters. The specific properties are selected by the users and can vary between executions. This simple concept turns out to be the basis in complex algorithms from many application areas, including sequence analysis and genotyping in bio-informatics, medical imaging, antimicrobial activity, market research, social networking etc. However, as the data volume continues to increase, the performance of clustering algorithms is heavily influenced by the memory subsystem, which is especially critical in real-time data analysis.

In this paper, we propose a novel and efficient implementation of Lloyd's k-means clustering algorithm to substantially reduce data movement along the memory hierarchy. Our contributions are based on the fact that the vast majority of processors are equipped with powerful Single Instruction Multiple Data (SIMD) instructions that are, in most cases, underused. SIMD improves the CPU computational power and, if used wisely, can be seen as

an opportunity to improve on the application data transfers by compressing/decompressing the data, specially for memory-bound applications. Our contributions include a SIMD-friendly data-layout organization, in-register implementation of key functions and SIMD-based compression. We demonstrate that using our optimized SIMD-based compression method, it is possible to improve the performance and energy of k-means by a factor of 4.5x and 8.7x respectively for a i7 Haswell machine, and 22x and 22.2x for Xeon Phi: KNL, running a single thread.

Retrospective View

This paper demonstrated the importance of vectorization in future HPC systems. Vectorization can turn CPU bound applications into memory bound, leaving more idle time to compress-decompress, specially when the processor runs out of reservation stations or load/store queue entries and it is unable to improve on memory-level parallelism. We have shown that integration of compression is feasible, as long as we can do it in a reasonable time. While the idea seemed to be a promising solution to make an efficient implementation of Lloyd's k-means algorithm, the experimental evaluations could have been improved by making an extensive comparison of our implementations with other parallel k-means implementations.

Roles of the Authors

I came up with the concept, performed preliminary investigation, planned which experiments to be carried out and implemented and conducted the planned experiments on multi-core systems. Cebrián helped us to get access to many-core system (i.e. KNL co-processor) and executed planned experiments on many-core system.

The first draft of the paper was prepared by me. Cebrián provided valuable comments which improved the overall quality of the paper. Natvig reviewed the paper and contributed to improve the presentation, organization and language of the paper.

4.2.6 C.2

Energy Efficiency Effects of Vectorization in Data Reuse Transformations for Many-core Processors – A Case Study
A. Al Hasib, L. Natvig, P. G. Kjeldsberg and J. M. Cebrián
Journal of Low Power Electronics and Applications
2017

Abstract

Thread-level and data-level parallel architectures have become the design of choice for a new era of energy efficient computing systems. However, these architectures have substantially higher requirements on the memory subsystem than scalar architectures, making memory latency and bandwidth critical in their overall efficiency. Data reuse exploration aims at reducing the pressure on the memory subsystem by exploiting the temporal locality in data accesses. In this paper, we investigate the effects on performance and energy efficiency from a data reuse methodology combined with parallelization and vectorization in many-core processors. As a test case, a “full-search motion estimation" kernel is evaluated on an Intel® Core™ i7-4700K (Haswell) multi-core processor as well as on an Intel® Xeon Phi™ many-core processor (Knights Landing). The Energy Delay Product (EDP) is used as metric for evaluating energy efficiency. The initial results running a single thread of scalar implementation of data reuse transformations show that performance and energy efficiency can be improved by a factor of 3.3x and 10.1x respectively on the Haswell system. The SSE version achieves performance improvements of around 10x over the scalar, and 103x better EDP. These results improve by 10 to 15% when using data reuse techniques. Finally, the most optimized version using data reuse and AVX512 achieves a speedup of 35.7x and an EDP improvement of 1271.6x when running a single thread on the Xeon Phi system.

Retrospective View

Though this paper was an extension of paper A.2, there was still ample room for improvement. The limitations and shortcoming of the paper are primarily to be attributed to the use of only one demonstrator application. Hence, the methodology could as well be tested with other real-world applications. Additionally, the experimental evaluations on many-core platform (i.e. KNL co-processor) could have been improved by evaluating all the different kernel implementations that are also considered for multi-core platforms.

Roles of the Authors

I carried out the preliminary investigation and came up with the initial idea. The initial idea was refined further through extensive discussions with Natvig and Kjeldsberg. Next I planned which experiments to be carried out, conducted the planned experiments and analyzed the obtained results. Cebrián helped us to get access to many-core system (i.e. KNL co-processor) in order to execute the planned experiments.

The first draft of the paper was prepared by me. Natvig and Kjeldsberg read the draft thoroughly and provided valuable comments which improved the overall quality of the paper. Cebrián reviewed the paper as well as made significant improvements on the presentation, organization and language of the paper.

4.3 Other Publications

- **Paper D.1:** Abdullah Al Hasib, Nikita Nikitin and Lasse Natvig. Cost-Comfort Balancing in a Smart Residential Building with Bidirectional Energy Trading, 33rd International Performance, Computing, and Communication Conference (IPCCC 2014), Austin, USA, 5-7 December, 2014.

- **Paper D.2:** Abdullah Al Hasib, Nikita Nikitin and Lasse Natvig. Cost-Comfort Balancing in a Smart Residential Building with Bidirectional Energy Trading. In Sustainable Internet and ICT for Sustainability (SustainIT 2015), Madrid, Spain. 2015, pp. 1-6.

Chapter 5

Concluding Remarks

Lessons learnt and the contributions from this thesis are summarized in this chapter. Section 5.1 gives an overview of the contributions in relation to the research questions and challenges put together earlier in this thesis. Finally, Section 5.2 lays down some future prospects as a direct consequence of this work.

5.1 Conclusion

In this thesis we have analyzed different approaches to improve performance and energy efficiency of a set of benchmarks and smart grid applications on multi-core and many-core systems. Though there are several different popular programming techniques for multi-core systems, we have primarily focused on vectorization techniques. These techniques showed very promising results in our initial energy efficiency studies.

In the first part of this research we have studied Black-Scholes, FFTW and Matrix multiplications to understand how vectorization techniques can improve energy efficiency of these well-known benchmarks. This study has demonstrated that vectorization can lead to a significant improvement on the application's energy efficiency. For instance, the AVX variant of Black-Scholes is $\approx$7x more on-chip energy efficient than the corresponding scalar variant.

In the next part, we have focused on smart grid applications, k-means clustering algorithms and a motion estimation kernel. We have observed that auto-vectorization performs rather poorly for these applications. For instance, the performance gain using SSE vectorization for Lloyd's k-means

clustering algorithm barely reached 30% using auto-vectorization. In contrast to the auto-vectorization, hand-tuned vectorization provided much better results in compute bound applications, like the evaluated full-search motion estimation kernel. However, for the time-series kernel, even the hand-tuned vectorization resulted in no performance benefit. In that scenario, the application turned into memory/cache bound due to simultaneous accesses of multiple data elements. Data compression can be a promising technique to address this limitation by reducing data movement across the memory hierarchy and expanding the effective cache capacity with little computational overhead. In the final part of this dissertation, we address this question by analyzing the compressibility of several real-world applications on several Intel multi-core and many-core platforms. We have shown that SIMD compression could in fact be beneficial to many real-world applications, particularly cache/memory bound applications.

The contributions of this study spread across different papers are here grouped together by the research questions they address. Some of the papers address multiple research questions, while most of the research questions are addressed in more than one paper.

5.1.1 RQ 1: To what extent is multi-core programming suitable for the selected smart grid applications?

Paper A.1, A.2, A.3 and B.1 shed light on this research question.

- **Paper A.1:** In paper A.1, we performed initial case-studies to demonstrate the impact of performance and power efficiency of vectorization and task-based programming in the OmpSs environment. We demonstrated that a significant improvement in performance and on-chip energy efficiency can be achieved using vectorization while the performance improvement from thread parallelism using OmpSs does not necessarily imply a better energy efficiency. We also found that the energy efficiency varies with problem size of the selected application kernels. This variation of energy efficiency with task size suggests that energy-aware task scheduling may adapt task sizes for energy efficient execution, which provides an interesting direction for future research.

- **Paper A.2:** In paper A.2, we presented a method to combine data reuse transformations with the OpenMP parallel programming model, so as to improve performance and energy efficiency of a motion estimation kernel on multi-core processors.

- **Paper A.3:** We further demonstrated in Paper A.3 that standard kernel libraries such as Intel MKL (Math Kernel Library), ATLAS (Automatically Tuned Linear Algebra Software) and LAPACK (Linear Algebra PACKage) can be used to boost up the performance of compute-intensive applications using data cleansing [88] as a demonstrator application. Additionally, MKL appeared to be more energy efficient than ATLAS or PLASMA (Parallel Linear Algebra Software for Multicore Architectures) [112] libraries on our test platforms even though the performance of these libraries were comparable.

- **Paper B.1:** After the good results shown in Paper A.1 and A.2, we decided to test the applicability of the vectorization on time series compute kernel provided by Powel AS (Paper B.1). However, we observed that simply using vectorization turned the kernel into memory bound, and had a negative impact on the overall performance. The processing units were often waiting for data to arrive from the memory subsystem. Thus, we uncovered that cache/memory bandwidth can be a bottleneck for an application when vectorization is applied, particularly for memory bound applications unless it is applied carefully.

5.1.2 RQ 2: To what extent can we improve the energy efficiency of the selected applications by using SIMD compression techniques?

- **Paper B.1:** The novelty of this work was to improve the performance and energy efficiency of a memory bound compute kernel (e.g. the Powel Time series compute kernel) by using a SIMD based differential compression method. Our strategy was to increase the cache block utilization and to reduce the total number of off-chip memory accesses by using V-PFORDelta, a lightweight real-time SIMD-based compression/decompression method based on a hybrid data structure for the compressed data (AoS[1] + SoA[2], aka AoSoA[3]).

5.1.3 RQ 3: Can we extend our research results to another application domain?

In papers C.1 and C.2, we extended our experiments into two different application domains to evaluate the effectiveness of vector processing and optimization. While C.1 paper contains the study of k-means - a widely used

[1] Array of Structures.
[2] Structure of Arrays.
[3] Array of Struct of Arrays OR Tiled Array of Structs

algorithm in data mining field, C.2 paper contains the study of a motion estimation algorithm - a multimedia application kernel.

- **Paper C.1:** We make an efficient implementation of a state-of-the-art k-means algorithm by using a SIMD-friendly data layout and by applying SIMD vectorization to make good use of the SIMD features available in modern architectures. Currently we have tested our proposal using AVX512, but ARM is about to release SVE [24], with support for up to 2048-bit vectors. This shows the importance of vectorization in future HPC systems. However, vectorization can also turn CPU bound applications into memory bound. In such scenario, it will leave more CPU idle time for data compression-decompression. Here, we have shown that integration of compression is feasible, as long as we can do it in a reasonable time.

- **Paper C.2:** The use of data reuse transformations together with vectorization is a promising approach to improve the performance and energy efficiency of massively parallel data-dominated applications (such as motion estimation) on multi- and many-core systems. Significant energy improvements can be achieved from throughput-oriented architectures that rely on low-power processing cores (e.g. KNL cores), especially if those cores provide SIMD/vector capabilities. These architectures have better energy efficiency (simple cores with low clock frequency) than complex cores available in commodity CPUs.

  As compared to multithreaded parallelism, data-parallelism through vector processing results in better energy savings even at peak core frequency. While doubling the number of cores results in approximately double the average power dissipated by the CPU, using vector units in Intel comes almost "for free" in terms of average power. Similar results have also been reported for several Intel and ARM CPUs in [113]. As a consequence, vector processing can be an attractive solution to improve energy efficiency without sacrificing performance, especially in a situation where performance trade-off is not desirable.

  The deployment order of different optimization techniques has a great impact on the application performance. First, we apply multithreading to exploit explicit parallelism across multiple cores, which is followed by fine-grained parallelism through vectorization at each core. Finally, data reuse transformations are applied as it depends on both multithreading and vectorization for further improvement. However, on applications that face scalability issues, users may want to limit

the amount of threads running in their application and rely more on SIMD units, since the energy cost of running on extra physical cores is much higher than using SIMD instructions.

5.2 Future Work

The foundation built in this research provides the opportunity to further explore and conduct experiments in other related directions. The following are a few of the possible fruitful extensions of this research work.

- In our research, we found that the energy efficiency varies with problem size of the selected application kernels. This variation of energy efficiency with task size suggests that energy-aware task scheduling may adapt task sizes for energy efficient execution, which provides an interesting direction for future research. It would be also interesting to extend our work by studying the impact of varying CPU clock frequencies, task scheduling policies, and using Turbo Boost Technology.

- In k-means optimization, our initial study shows that the use of precomputed values using a look-up table can lead to more than 30% performance improvement for the single threaded SSE vectorized k-means kernel. This can be explored more extensively to understand the effect of using a look-up table while approximating the distance vector for determining the membership of a data point.

- Furthermore, our evaluation of k-means optimization was limited to only a couple of data-sets. It would be also interesting to study our proposed method using additional input sets with varying sizes and compression ratios, to get a better idea of the potential of the proposal in different fields of application. This work can be extended by studying the effectiveness of SIMD compression method in other types of data-structures (e.g. B-Trees) or algorithms as well.

- Our SIMD optimization of motion estimation kernel study was only limited to static and dynamic scheduling (in the KNL co-processor). Therefore, this work can be further extended to analyze the effect of using more advanced scheduling method (e.g., guided scheduling) along with compiler assisted selected lock assignment on the data reuse transformations in the simultaneous multithreading environment. Also extending our study by running the experiments on an execution platform supporting a concept like drowsy cache that powers down

the unused parts of the cache would be interesting as it will give more comparable results against the results of Wuytack et al [114].

Bibliography

[1] B. H. Calhoun, S. Khanna, R. Mann and J. Wang. 'Sub-threshold Circuit Design with Shrinking CMOS Devices'. In: *IEEE International Symposium on Circuits and Systems*. May 2009, pp. 2541–2544. DOI: 10.1109/ISCAS.2009.5118319.

[2] Vivek De. 'Energy-Efficient Computing in Nanoscale CMOS'. In: *IEEE Design Test* 33.2 (Apr. 2016), pp. 68–75. ISSN: 2168-2356.

[3] Shekhar Borkar and Andrew A. Chien. 'The Future of Microprocessors'. In: *Communications of the ACM* 54.5 (May 2011), pp. 67–77. ISSN: 0001-0782.

[4] P. Chaparro, J. Gonzalez, G. Magklis, Cai Qiong and A. Gonzalez. 'Understanding the Thermal Implications of Multi-Core Architectures'. In: *IEEE Transactions on Parallel and Distributed Systems* 18.8 (Aug. 2007), pp. 1055–1065. ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.1092.

[5] David J. Brown and Charles Reams. 'Toward Energy-efficient Computing'. In: *Communications of the ACM* 53.3 (Mar. 2010), pp. 50–58. ISSN: 0001-0782.

[6] Karl Rupp. *40 Years of Microprocessor Trend Data*. URL: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/.

[7] G. E. Moore. 'Cramming More Components onto Integrated Circuits'. In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[8] Intel. *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors*. Tech. rep. Nov. 2008.

[9] Luca Benini and Giovanni De Micheli. '"System-level Power Optimization: Techniques and Tools"'. In: *ACM Transaction on Design Automation of Electronic Systems* 5.2 (Apr. 2000), pp. 115–192.

[10] V. Raghunathan, M. B. Srivastava and R. K. Gupta. 'A Survey of Techniques for Energy Efficient On-chip Communication'. In: *IEEE Design Automation Conference*. June 2003, pp. 900–905.

[11] Parthasarathy Ranganathan. 'Recipe for Efficiency: Principles of Power-Aware Computing'. In: *Communication of the ACM* 53.4 (Apr. 2012).

[12] R. H. Dennard, F. H. Gaensslen, H. N. Yu, V. Leo Rideovt, E. Bassous and A. R. Leblanc. 'Design of Ion-implanted MOSFET's with Very Small Physical Dimensions'. In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (Oct. 2007), pp. 38–50. ISSN: 1098-4232.

[13] Cor Meenderinck and Ben Juurlink. '(When) Will CMPs Hit the Power Wall?' In: *Proceedings of the Euro-Par Workshops - Parallel Processing*. Springer-Verlag, 2009, pp. 184–193.

[14] W. Huang, K. Rajamani, M. R. Stan and K. Skadron. 'Scaling with Design Constraints: Predicting the Future of Big Chips'. In: *IEEE Micro* 31.4 (July 2011), pp. 16–29. ISSN: 0272-1732. DOI: 10.1109/MM.2011.42.

[15] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam and Doug Burger. 'Dark Silicon and the End of Multicore Scaling'. In: *Proceedings of the International Symposium on Computer Architecture*. ISCA '11. New York, NY, USA, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6.

[16] Manuel F. Dolz, Francisco D. Igual, Thomas Ludwig, Luis Pinuel and Enrique S. Quintana-Ortí. 'Balancing Task- and Data-level Parallelism to Improve Performance and Energy Consumption of Matrix Computations on the Intel Xeon Phi'. In: *Computers and Electrical Engineering* 46.C (Aug. 2015), pp. 95–111. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2015.06.009.

[17] Barbara Chapman, Gabriele Jost and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN: 9780262533027.

[18] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell and Judit Planas. 'OmpSs - A Proposal for Programming Heterogeneous Multi-core Architetcures'. In: *Parallel Processing Letters* 21 (Mar. 2011), pp. 173–193.

[19] C. E. Leiserson. 'The Cilk++ Concurrency Platform'. In: *Proceedings of the Design Automation Conference*. July 2009, pp. 522–527. DOI: 10.1145/1629911.1630048.

[20] W. Kim and M. Voss. 'Multicore Desktop Programming with Intel Threading Building Blocks'. In: *IEEE Software* 28.1 (Jan. 2011), pp. 23–31. ISSN: 0740-7459.

[21] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124159884.

[22] Juan M. Cebrián, Lasse Natvig and Jan Christian Meyer. 'Performance and Energy Impact of Parallelization and Vectorization Techniques in Modern Microprocessors'. In: *Computing* (2013). ISSN: 1436-5057.

[23] R. James. *Additional AVX-512 Instructions*. 2014. URL: https://software.intel.com/en-us/blogs/additional-avx-512-instructions.

[24] Nigel Stephens. *Technology Update: The Scalable Vector Extension (SVE) for the ARMv8-A architecture*. 2016. URL: https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture.

[25] Muhammad Shafique, Siddharth Garg, Tulika Mitra, Sri Parameswaran and Jorg Henkel. 'Dark Silicon As a Challenge for Hardware/Software Co-design: Invited Special Session Paper'. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. CODES '14. New Delhi, India: ACM, 2014, 13:1–13:10. ISBN: 978-1-4503-3051-0. DOI: 10.1145/2656075.2661645.

[26] Dustin Feld, Thomas Soddemann, Michael Junger and Sven Mallach. 'Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs'. In: *Proceedings of the International Workshop on Code Optimisation for Multi and Many Cores*. COSMIC '15. San Francisco Bay Area, CA, USA: ACM, 2015, 2:1–2:10. ISBN: 978-1-4503-3316-0. DOI: 10.1145/2723772.2723776.

[27] Lasse Natvig and Alexandru C. Iordan. 'Green Computing: Saving Energy by Throttling, Simplicity and Parallelization'. In: *CEPIS Upgrade* 12.4 (Oct. 2011), pp. 49–58. ISSN: 1684-5285.

[28] G. Lawson, M. Sosonkina and Y. Shen. 'Energy Evaluation for Applications with Different Thread Affinities on the Intel Xeon Phi'. In: *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshop*. Oct. 2014, pp. 54–59.

[29] Davendar Kumar Ojha and Geeta Sikka. 'A Study on Vectorization Methods for Multicore SIMD Architecture Provided by Compilers'. In: *Advances in Intelligent Systems and Computing* 248.1 (2014), pp. 723–728. ISSN: 978-3-319-03107-1.

[30] Juan M. Cebrián, Magnus Jahre and Lasse Natvig. 'Optimized Hardware for Suboptimal Software: The Case for SIMD-aware Benchmarks'. In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. Mar. 2014, pp. 66–75.

[31] Avinash Sodani. *Race to Exascale: Challenges and Opportunities*. URL: http://www.microarch.org/micro44/files/Micro%20Keynote%20Final%20-%20Avinash%20Sodani.pdf.

[32] Hans Michael Gerndt, Michael Glaß, Sri Parameswaran and Barry L. Rountree. 'Dark Silicon: From Embedded to HPC Systems (Dagstuhl Seminar 16052)'. In: *Dagstuhl Reports* 6.1 (2016). Ed. by Hans Michael Gerndt, Michael Glaß, Sri Parameswaran and Barry L. Rountree, pp. 224–244. ISSN: 2192-5283. DOI: 10.4230/DagRep.6.1.224.

[33] A. Pahlevan, J. Picorel, A. P. Zarandi, D. Rossi, M. Zapater, A. Bartolini, P. G. Del Valle, D. Atienza, L. Benini and B. Falsafi. 'Towards Near-Threshold Server Processors'. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 7–12.

[34] J. Henkel, S. Pagani, H. Khdr, F. Kriebel, S. Rehman and M. Shafique. 'Towards Performance and Reliability-Efficient Computing in the Dark Silicon Era'. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 1–6.

[35] Bharathwaj Raghunathan, Yatish Turakhia, Siddharth Garg and Diana Marculescu. 'Cherry-picking: Exploiting Process Variations in Dark-silicon Homogeneous Chip Multi-processors'. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. Grenoble, France, 2013, pp. 39–44. ISBN: 978-1-4503-2153-2.

[36] Qiang Liu and Wayne Luk. 'Heterogeneous Systems for Energy Efficient Scientific Computing'. In: *Proceedings of International Symposium on Reconfigurable Computing: Architectures, Tools and Applications*. Ed. by Oliver C. S. Choy, Ray C. C. Cheung, Peter Athanas and Kentaro Sano. Springer Berlin Heidelberg, 2012, pp. 64–75. ISBN: 978-3-642-28365-9.

[37] Lei Yang, Weichen Liu, Weiwen Jiang, Chao Chen, Mengquan Li, Peng Chen and Edwin H.M. Sha. 'Hardware-software Collaboration for Dark Silicon Heterogeneous Many-core Systems'. In: *Future Generation Computer Systems* 68 (2017), pp. 234–247. ISSN: 0167-739X. DOI: http://doi.org/10.1016/j.future.2016.09.012.

[38] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao and Doug Burger. 'A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services'. In: *Proceeding of the International Symposium on Computer Architecuture*. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 13–24. ISBN: 978-1-4799-4394-4.

[39] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald and R. Menon. *Parallel Programming in OpenMP*. 2001.

[40] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan and G. Zhang. 'The Design of OpenMP Tasks'. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (Mar. 2009), pp. 404–418.

[41] Judit Planas, Rosa M. Badia, Eduard Ayguadé and Jesus Labarta. 'Hierarchical Task-based Programming With StarSs'. In: *The International Journal of High Performance Computing Applications* 23.3 (2009), pp. 284–299. DOI: 10.1177/1094342009106195.

[42] *The nanos group site: The mercurium compiler*. URL: http://nanos.ac.upc.edu/mcxx.

[43] Thomas Willhalm and Nicolae Popovici. 'Putting Intel Threading Building Blocks to Work'. In: *Proceedings of the International Workshop on Multicore Software Engineering*. IWMSE '08. Leipzig, Germany: ACM, 2008, pp. 3–4. ISBN: 978-1-60558-031-9. DOI: 10.1145/1370082.1370085.

[44] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. *MPI – The Complete Reference*. 2nd. London, England: The MIT press, 1998. ISBN: 9780124159334.

[45] Charles E. Leiserson. 'The Cilk++ concurrency platform'. In: *The Journal of Supercomputing* 51.3 (2010), pp. 244–257. ISSN: 1573-0484. DOI: 10.1007/s11227-010-0405-3.

[46] Umut A. Acar, Arthur Chargueraud and Mike Rainey. 'Scheduling Parallel Programs by Work Stealing with Private Deques'. In: *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 219–228. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442538.

[47] J. E. Stone, D. Gohara and G. Shi. 'OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems'. In: *Computing in Science Engineering* 12.3 (May 2010), pp. 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69.

[48] J. Fang, A. L. Varbanescu and H. Sips. 'A Comprehensive Performance Comparison of CUDA and OpenCL'. In: *Proceedings of the International Conference on Parallel Processing*. Sept. 2011, pp. 216–225.

[49] S.J. Pennycook, S.D. Hammond, S.A. Wright, J.A. Herdman, I. Miller and S.A. Jarvis. 'An investigation of the performance portability of OpenCL'. In: *Journal of Parallel and Distributed Computing* 73.11 (2013). Novel architectures for high-performance computing, pp. 1439–1450. ISSN: 0743-7315. DOI: http://doi.org/10.1016/j.jpdc.2012.07.005.

[50] D.J.N. van Doets H.C.; Eijck. 'The Haskell Road to Logic, Maths and Programming, Second Edition'. In: *Texts in Computing* 4 (2012), pp. 1–34. ISSN: 978-0-9543006-9-2.

[51] Kevin Hammond. 'Glasgow Parallel Haskell (GpH)'. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 768–779. ISBN: 978-0-387-09766-4.

[52] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell and Vinod Grover. 'Accelerating Haskell Array Codes with Multicore GPUs'. In: *Proceedings of the Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. Austin, Texas, USA: ACM, 2011, pp. 3–14. ISBN: 978-1-4503-0486-3. DOI: 10.1145/1926354.1926358.

[53] Silvia Breitinger, Ulrike Klusik and Rita Loogen. 'From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View'. In: *Proceedings of International Symposium on Principles of Declarative Programming*. Ed. by Catuscia Palamidessi, Hugh Glaser and Karl Meinke. Springer Berlin Heidelberg, 1998, pp. 318–334. ISBN: 978-3-540-49766-0.

[54] Jeff Epstein, Andrew P. Black and Simon Peyton-Jones. 'Towards Haskell in the Cloud'. In: *SIGPLAN Not.* 46.12 (Sept. 2011), pp. 118–129. ISSN: 0362-1340. DOI: 10.1145/2096148.2034690.

[55] Prabhat Totoo and Hans-Wolfgang Loid. 'Parallel Haskell implementations of the N-body problem'. In: *Concurrency and Computation - Practice and Experience* 72.0 (2012), pp. 1–34. ISSN: 0743-7315.

[56] Daniel Kusswurm. 'Streaming SIMD Extensions'. In: *Modern X86 Assembly Language Programming*. Apress, Berkeley, CA, Nov. 2014, pp. 179–206. ISBN: 978-1-4842-0064-3.

[57] Varghese George, Sanjeev Jahagirdar, Chao Tong, K. Smits, Satish Damaraju, S. Siers, Ves Naydenov, Tanveer Khondker, Sanjib Sarkar and Puneet Singh. 'Penryn: 45-nm next generation Intel® Core™ 2 processor'. In: *Proceedings of the Asian Solid-State Circuits Conference*. Nov. 2007, pp. 14–17. DOI: 10.1109/ASSCC.2007.4425784.

[58] N. Kurd, J. Douglas, P. Mosalikanti and R. Kumar. 'Next Generation Intel Micro-architecture (Nehalem) Clocking Architecture'. In: *Proceedings of the Symposium on VLSI Circuits*. June 2008, pp. 62–63. DOI: 10.1109/VLSIC.2008.4585952.

[59] Intel. *Intel Advanced Vector Extensions Programming Reference*. June 2011. URL: http://software.intel.com/file/36945.

[60] Paul Cockshott and Kenneth Renfrew. 'SIMD Programming in Assembler and C'. In: *SIMD Programming Manual for Linux and Windows*. London: Springer London, 2004, pp. 23–46. ISBN: 978-1-4471-3862-4. DOI: 10.1007/978-1-4471-3862-4_3.

[61] Intel. *Intel Intrinsics Guide*. URL: http://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[62] Agner Fog. *VCL: C++ Vector Class Library*. URL: http://www.agner.org/optimize/vectorclass.pdf.

[63] Clark L. Coleman. *Using Inline Assembly With gcc*. Nov. 2011. URL: http://www.mujweb.cz/tvrzsky/asm/gcc-inline-asm.pdf.

[64] M. A. Nichols, H. J. Siegel and H. G. Dietz. 'Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler'. In: *IEEE Transactions on Parallel and Distributed Systems* 4.2 (Feb. 1993), pp. 222–234. ISSN: 1045-9219. DOI: 10.1109/71.207596.

[65] Jaekyu Lee, Hyesoon Kim and Richard Vuduc. 'When Prefetching Works, When It Doesn't, and Why?' In: *ACM Transactions on Architecture and Code Optimization* 9.1 (Mar. 2012), pp. 1–29.

[66] Keith E Mathias and L. Darrell Whitley. 'Changing Representations During Search: A Comparative Study of Delta Coding'. In: *Evolutionary Computation* 2.3 (1994), pp. 249–278.

[67] Hugh E. Williams and Justin Zobel. 'Compressing Integers for Fast File Access'. In: *The Computer Journal* 42.3 (1999), pp. 192–201. DOI: 10.1093/comjnl/42.3.193.

[68] PG Howard and JS Vitter. 'Arithmetic Coding for Data Compression'. In: *Proceedings of the IEEE* 82.6 (1994), pp. 857–865. ISSN: 0018-9219. DOI: 10.1109/5.286189.

[69] Jiangong Zhang, Xiaohui Long and Torsten Suel. 'Performance of Compressed Inverted List Caching in Search Engines'. In: *Proceedings of the International Conference on World Wide Web*. Apr. 2008, pp. 387–396. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367550.

[70] Jonathan Goldstein, Raghu Ramakrishnan and Uri Shaft. 'Compressing Relations and Indexes'. In: *Proceedings of the International Conference on Data Engineering*. Feb. 1998, pp. 370–379.

[71] Benjamin Schlegel, Rainer Gemulla and Wolfgang Lehner. 'Fast Integer Compression using SIMD Instructions'. In: *International Workshop on Data Management on New Hardware*. June 2010, pp. 34–40.

[72] Alexander A. Stepanov and Anil R. Gangolli. 'SIMD Based Decoding of Posting Lists'. In: *Proceedings of the International Conference on Information and Knowledge Management*. Oct. 2011, pp. 317–326.

[73] Naiyoung Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu and Sheng Lin. 'Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units'. In: *Proceedings of the VLDB Endowment*. Vol. 4. Sept. 2011, pp. 470–481.

[74] Xudong Zhang, Wayne Xin Zhao, Dongdong Shan and Hongfei Yan. 'Group-Scheme: SIMD-based Compression Algorithms for Web Text Data'. In: *Proceedings of the International Conference on BigData*. Oct. 2013, pp. 525–530. ISBN: 978-1-4799-1292-6.

[75] D. Lemire and L. Boytsov. 'Decoding Billions of Integers Per Second through Vectorization'. In: *Software: Practice and Experience* 45 (Jan. 2015), pp. 1–29.

[76] Daniel Lemire, Leonid Boytsov and Nathan Kurz. 'SIMD Compression and the Intersection of Sorted Integers'. In: *Software: Practice and Experience* (Apr. 2015).

[77] Jack J. Dongarra, Piotr Luszczek and Antoine Petitet. 'The LINPACK Benchmark: Past, Present and Future'. In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), pp. 803–820. ISSN: 1532-0634. DOI: 10.1002/cpe.728.

[78] Matteo Frigo and Steven G. Johnson. 'The Design and Implementation of FFTW3'. In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[79] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li. ' The PARSEC Benchmark Suite - Characterization and Architectural Implications'. In: *Proceedings of the Conference on International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. 2008, pp. 72–81.

[80] J. Planas, R. M. Badia, E. Ayguade and J. Labarta. 'Self-Adaptive OmpSs Tasks in Heterogeneous Environments'. In: *International Symposium on Parallel and Distributed Processing*. May 2013, pp. 138–149. DOI: 10.1109/IPDPS.2013.53.

[81] Fischer Black and Myron S Scholes. 'The Pricing of Options and Corporate Liabilities'. In: *Journal of Political Economy* 81.3 (1973), pp. 637–54.

[82] *Top500 Supercomputer Sites*. URL: http://www.top500.org.

[83] *BLAS (Basic Linear Algebra Subprograms)*. URL: http://www.netlib.org/blas/.

[84] R. Clint Whaley and Jack J. Dongarra. 'Automatically Tuned Linear Algebra Software'. In: *Proceedings of the International Conference on Supercomputing*. SC '98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27. ISBN: 0-89791-984-X. URL: http://dl.acm.org/citation.cfm?id=509058.509096.

[85] *The Power of Powel*. URL: http://www.powel.com/About-Powel/.

[86] Ole Martin Sørli and Magne Tøndel. 'Efficient multicore programming for industrial applications'. MA thesis. Norwegian University of Science and Technology (NTNU), June 2010.

[87] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook*. Springer, 2010. ISBN: 978-0-387-09822-7.

[88] J. Chen, W. Li, A. Lau, J. Cao and K. Wang. 'Automated Load Curve Data Cleansing in Power Systems'. In: *IEEE Transactions on Smart Grid* 1.2 (Sept. 2010), pp. 213–221. ISSN: 1949-3053. DOI: 10.1109/TSG.2010.2053052.

[89] S. Lloyd. 'Least Squares Quantization in PCM'. In: *IEEE Transaction Information Theory* 28.2 (Sept. 2006), pp. 129–137. ISSN: 0018-9448.

[90] David Arthur and Sergei Vassilvitskii. 'K-means++: The Advantages of Careful Seeding'. In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*. 2007, pp. 1027–1035. ISBN: 978-0-898716-24-5.

[91] Greg Hamerly. 'Making K-means Even Faster'. In: *Proceedings of the International Conference on Data Mining*. Apr. 2010, pp. 130–140.

[92] Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao and Long Gao. 'A Vectorized K-means Algorithm for Intel Many Integrated Core Architecture'. In: *International Symposium on Advanced Parallel Processing Technologies*. Aug. 2013, pp. 277–294. ISBN: 978-3-642-45292-5.

[93] Ali Hadian and Saeed Shahrivari. 'High Performance Parallel K-means Clustering for Disk-resident Datasets on Multi-core CPUs'. In: *The Journal of Supercomputing* 69.2 (2014), pp. 845–863.

[94] Mario Zechner and Michael Granitzer. 'K-Means on the Graphics Processor: Design And Experimental Analysis'. In: *International Journal on Advances in Systems and Measurements* 2.3 (2009), pp. 224–235. ISSN: 1942-261x.

[95] J. Mathew and R. Vijayakumar. 'Enhancement of Parallel K-means Algorithm'. In: *Proceedings of the International Conference on Innovations in Information, Embedded and Communication Systems*. Mar. 2015, pp. 1–6.

[96] Hari Kalva, Aleksandar Colic, Adriana Garcia and Borko Furht. 'Parallel Programming for Multimedia Applications'. In: *Multimedia Tools Applications* 51.2 (2011), pp. 801–818.

[97] A. Sodani. 'Knights landing (KNL) - 2nd Generation Intel Xeon Phi processor'. In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. Aug. 2015, pp. 1–24. DOI: 10.1109/HOTCHIPS.2015.7477467.

[98] *Performance Application Programming Interface*. URL: http://icl.cs.utk.edu/papi/index.html.

[99] Jan Edler. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. URL: http://www.cs.wisc.edu/~markhill/DineroIV.

[100] Valgrind. *Lackey: An Example Tool*. URL: http://valgrind.org/docs/manual/lk-manual.html.

[101] Hallgeir Lien, Lasse Natvig, Abdullah Al Hasib and Jan Christian Meyer. 'Case Studies of Multi-core Energy Efficiency in Task Based Programs'. In: *Proceedings of the International Conference on ICT as Key Technology against Global Warming*. Vol. 7453. Sept. 2012, pp. 44–54.

[102] Johannes Hofmann, Jan Eitzinger and Dietmar Fey. 'Execution-Cache-Memory Performance Model: Introduction and Validation'. In: *CoRR* abs/1509.03118 (2015). URL: http://arxiv.org/abs/1509.03118.

[103] Intel. *Intel Energy Checker*. 2010. URL: https://software.intel.com/sites/default/files/m/6/6/4/5/0/Intel_R__Energy_Checker_SDK--Companion_Applications_User_Guide.pdf.

[104] Abdullah Al Hasib, Per Gunnar Kjeldsberg and Lasse Natvig. 'Performance and Energy Efficiency Analysis of Data Reuse Transformation Methodology on Multicore Processor'. In: *Proceedings of the Euro-Par 2012: Parallel Processing Workshops*. Vol. 7640. LNCS. 2013, pp. 337–346. ISBN: 978-3-642-36949-0.

[105] Abdullah Al Hasib and Lasse Natvig. 'Performance Optimization and Evaluation of a Data Cleansing Algorithm on Multicore Processors'. In: *The 9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*. Academia Press, 2013. ISBN: 9789038221908.

[106] Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig. 'V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series'. In: *22nd International Conference on High Performance Computing*. Dec. 2015, pp. 44–54.

[107] Amanda Svensson Marvel. *Memory Layout Transformations*. 2013. URL: https://software.intel.com/en-us/articles/memory-layout-transformations.

[108] Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig. 'A Vectorized k-means Algorithm for Compressed Datasets: Design and Experimental Analysis'. In: *Proceedings of the International Conference on High Performance Computing*. May 2017, pp. 1–10.

[109] Abdullah Al Hasib, Lasse Natvig, Per Gunnar Kjeldberg and Juan M. Cebrián. 'Energy Efficiency Effects of Vectorization in Data Reuse Trans- formations for Many-core Processors - A Case Study'. In: *Journal of Low Power Electronics and Applications* (May 2017).

[110] A. A. Hasib, N. Nikitin and L. Natvig. 'Load Scheduling in Smart Buildings with Bidirectional Energy Trading'. In: *Proceedings of the IEEE International Performance Computing and Communications Conference*. Dec. 2014, pp. 1–2.

[111] A. A. Hasib, N. Nikitin and L. Natvig. 'Cost-comfort Balancing in a Smart Residential Cuilding with Bidirectional Energy Trading'. In: *Proceedings of the Conference on Sustainable Internet and ICT for Sustainability*. Apr. 2015, pp. 1–6.

[112] *PLASMA - Parallel Linear Algebra Software for Multicore Architectures*. URL: http://icl.cs.utk.edu/plasma/.

[113] Juan M. Cebrián, Magnus Jahre and Lasse Natvig. 'ParVec: Vectorizing the PARSEC Benchmark Suite'. In: *Computing* 97.11 (2015), pp. 1077–1100. ISSN: 1436-5057.

[114] J.Ph. Diguet, S. Wuytack, F. Catthoor et al. 'Formalized Methodology for Data Reuse Exploration for Low-Power Hierarchical Memory Mappings'. In: *IEEE Transactions on VLSI Systems* 6 (1998), pp. 529–537.

Part II

Papers

Paper A.1

Case Studies of Multi-core Energy Efficiency in Task Based Programs

Hallgeir Lien, Lasse Natvig, Abdullah Al Hasib, and Jan Christian Meyer

2nd International Conference on ICT as Key Technology against Global Warming

Abstract

In this paper, we present three performance and energy case studies of benchmark applications in the OmpSs environment for task based programming. Different parallel and vectorized implementations are evaluated on an Intel® Core™ i7-2600 quad-core processor. Using FLOPS/W derived from chip MSR registers, we find AVX code to be clearly most energy efficient in general. The peak on-chip GFLOPS/W rates are: Black-Scholes (BS) 0.89, FFTW 1.38 and Matrix Multiply (MM) 1.97. Experiments cover variable degrees of thread parallelism and different OmpSs task pool scheduling policies. We find that maximum energy efficiency for small and medium sized problems is obtained by limiting the number of parallel threads. Comparison of AVX variants with non-vectorized code shows $\approx$ 6–7x (BS) and $\approx$ 3–5x (FFTW) improvements in on-chip energy efficiency, depending on the problem size and degree of multithreading.

1 Introduction

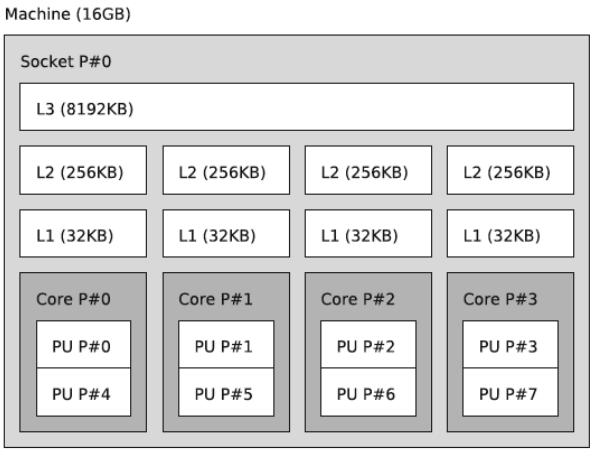
Saving energy is now a top priority in most computing systems. Sensor networks which report over long time frames are installed in environments where it is expensive or impossible to replace batteries. Mobile computing devices have severely restricted operation time without recharging. Computers produce less heat, less noise, and are cheaper to operate if they consume less energy.

Recently, we have seen a convergence between embedded systems and High Performance Computing. Both these market segments now have energy efficiency as a major design goal. The convergence is exemplified in the Mont Blanc project, which is part of the European Exascale Software Initiative (EESI). Mont Blanc aims at developing a European scalable and power efficient HPC platform based on low-power embedded technology [1].

The Green500 list ranks the world's most energy efficient supercomputers [2]. The ranking is based on the FLOPS/W metric for LINPACK and the top entry in the November 2011 list achieved 2.03 GFLOPS/W. Motivated by Mont Blanc targeting the Green500 list, we selected FLOPS/W as a metric for our studies.

Task Based Programming (TBP) has recently gained increasing interest. In some TBP systems the programmer must take care of all data dependencies between the tasks by explicit synchronizations. In newer, dependency aware TBP systems [3] the cumbersome synchronization is transferred to the run- time system. OmpSs uses this automatic run-time parallelization approach, and provides mechanisms for executing tasks on accelerators such as GPUs [4], thus simplifying the programming of heterogeneous and hybrid architectures. OmpSs will be used in the Mont Blanc project [5]. We have chosen the Black-Scholes benchmark and Matrix Multiply already implemented with OmpSs for our case studies. In addition, we adapted an OpenMP benchmark of FFTW for OmpSs. We implemented SSE and AVX vectorization for Black-Scholes and compiled FFTW without vectorization, with SSE and with AVX, while Matrix Multiply was already vectorized with AVX through its use of ATLAS [6].

This paper presents energy efficiency results for three benchmarks, comparing the effects of applying vectorization and thread parallelism. Problem sizes are restricted to minimize interactions with memory, isolating on-chip energy consumption. It is an initial effort toward understanding overall system power by examining incremental sets of subsystems.



Model #	42
Stepping	7
Clock frequency	1.6 - 3.4GHz
Physical cores	4
Logical cores	8
Main memory	16GB

Figure 1: Intel® Core™ i7-2600 Sandy Bridge multi-core processor architecture (left) and specification (right)

Our contributions are on-chip energy efficiency results for Black-Scholes, FFTW and matrix multiplication on the recent Intel Sandy Bridge architecture, and discussion of the relative benefits of parallelization and vectorization.

The paper is organized as follows: Section 2 describes the computer used in the experiments, motivates and defines the selection of energy efficiency metric, and introduces the selected benchmarks. Section 3 explains how we performed the energy measurements, and organized the experiments to achieve stable and reproducible results. We outline the vectorization and parallelizations, followed by a discussion of the main results. Section 4 describes related work, before the paper is concluded in Section 5.

2 Background

2.1 Execution Platform, SSE and AVX

All experiments were executed on a four core desktop computer that can execute 8 threads using Intel HyperthreadingTM. Its main architecture and specifications are shown in Figure 1.

All cores were clocked at their maximum rate of 3.4 GHz. Cache sizes, line sizes and associativity are described in Table 1. Latencies are taken from [7]. The Intel Sandy Bridge processors allow vectorization using SSE or AVX. AVX registers extend the 128 bit SSE registers with an additional 128

Table 1: Cache information for Intel Core i7-2600, 3.4GHz

Cache	Size	Sharing	Ways of Associativity	Line size	Latency (cycles)
Level 1 Instruction	32 KB	Private	8	64 B	4
Level 1 Data	32 KB	Private	8	64 B	4
Level 2	256 KB	Private	8	64 B	12
Level 3	8 MB	Shared	16	64 B	26-31

bits, and can theoretically double the throughput [8]. SSE and AVX are programmed using intrinsics, inline assembly, or automatic vectorization by the compiler.

2.2 Performance and Energy Metrics

There is a trade-off between the partly conflicting goals of high performance and low energy consumption. Comparing systems based on energy consumption alone would motivate the use of very slow processors with low frequency, since energy is the product of power and execution time. The *Energy-Delay Product (EDP)* places greater emphasis on performance, and corresponds to the reciprocal of performance pr. energy unit. Different metrics are appropriate to different cases when studying energy efficiency. Rivoire et al. [9] give a readable introduction to the pros and cons of various energy efficiency metrics. $Performance^N/Watt$ is among the most general, as it allows adjusting the balance between high performance and low energy consumption. N = 0 implies a focus on the power consumption alone, while N = 2 corresponds to EDP.

Any FLOPS performance metric implies a definition of how many floating point operations are required to handle a given problem size. One method would be to measure the number of operations per experiment, using performance counters. This would also count unnecessary operations, and be poorly suited to comparing performance between implementations. In this work, FLOPS rate was measured by counting or estimating the number of useful floating point operations and dividing by execution time. Integer operations such as bit-wise logical operations and shifts were ignored. Further details on the operation counts can be found in [10].

Energy measurements were obtained from the energy consumption fields of the non-architectural Machine State Registers (MSRs) made available by the Running Average Power Limit (RAPL) interface [11]. Because these values

only reflect chip level energy consumption, we observe the L3 miss rate to find the range of problem sizes where the application is being executed on-chip. As long as the L3 miss rate is close to zero, our on-chip energy measurements give a fair comparison of energy efficiency for the different implementations.

2.3 Selection of Benchmarks

Our choice of applications is motivated by the Mont Blanc project, leading to use of OmpSs, and benchmark selection from potential target applications [5]. Black-Scholes is part of the PARSEC Benchmark Suite for shared memory computers [12]. It calculates prices for a portfolio of European stock options by evaluating the Black-Scholes formula for each element of a data set. A financial market is modeled by repeating this computation over time. FFTW (Fastest Fourier Transform in the West) is a widely used FFT library. The FFTW library achieves high performance by automatically adapting its algorithm for the machine it is run on. It first creates a plan of execution for the given problem, and then executes it. A plan is created by heuristically tailoring execution to the current system (e.g. querying cache sizes), and several different plans are tested to find the fastest candidate. Measurement can be omitted to save plan creation time, when less efficient execution is acceptable [13]. The third application studied is Matrix Multiplication implemented with OmpSs. It creates tasks from multiplication tiles, calling BLAS gemm at the tile level. We use the ATLAS library for this, because of its AVX support.

3 Experiments and Results

3.1 Methods

We use the RAPL MSR interface to read out energy used by the processor chip. The bits 12:8 of the *MSR_RAPL_POWER_UNIT* register describe the granularity of the energy values. The default value is 2^{-16}J $\approx 15.3\mu$J. Consumed energy is read from the bits 31:0 of the *MSR_PKG_ENERGY_STATUS* register, which has a wrap- around time of about 60 seconds on high processor load [11]. Our experiments complete in a few seconds, remaining safely within this limit. Data access was kept within the multi-core chip by limiting problem sizes to fit in the last level cache (LLC). As the RAPL registers do not reflect the cost of off-chip memory, its magnitude is not visible in our results, making it necessary to restrict its influence. LLC miss rates were recorded using performance monitoring counters, in order to validate that predicted limits for on-chip problem sizes are correct. The

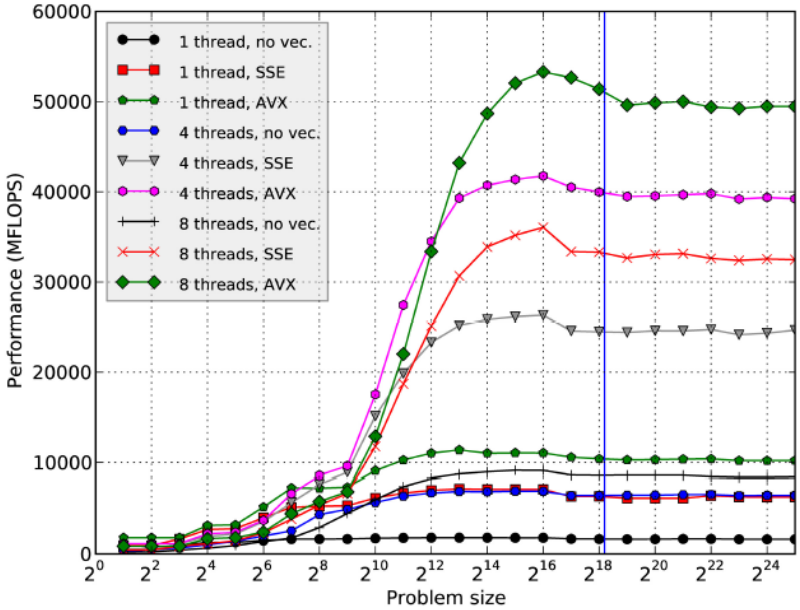


Figure 2: Performance vs. problem size for Black-Scholes. The vertical line marks the 8MB point, i.e. the problem size where application data require the entire LLC.

changes in application behavior observed at the LLC limit are visible in our performance results. Every experiment was run 10 times and we plot the median value for each problem size. The first sample points are discarded, in order to remove cache cold start effects. The results are reproducible and stable, with a relative standard deviation less than 3% for the relevant problem sizes. The standard deviations of runs are far smaller than the margins separating different implementations. All experiments were run under openSuse 11.4 (x86_64) running Linux kernel 2.6.37.6, and all OmpSs applications were compiled using *sscc* from the OmpSs package. As *sscc* translates at source level, *gcc* 4.7.0 generated the native code. Nanos++ runtime version 0.6a was used for all experiments.

3.2 Black-Scholes

Vectorization of Black-Scholes made it necessary to implement natural logarithm and exponential functions. We adapted code from the Cephes Mathematical Library [13]; further details can be found in H. Lien's Master thesis [10]. Black-Scholes uses 6 input- and one output-array, each containing N 32-bit floating point numbers, giving a memory footprint of 28^N bytes,

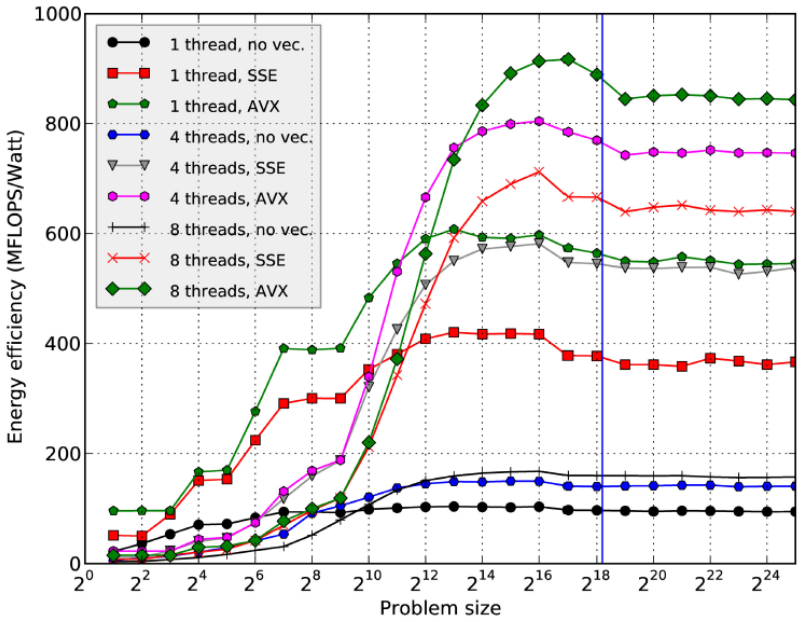


Figure 3: Energy efficiency vs. problem size for Black-Scholes. The vertical line marks the 8MB point, i.e. the problem size where application data require the entire LLC.

where N is the problem size. The largest problem that can fit the LLC is N $= 2^{18}$, as $2^{18} \cdot 28$ B $= 7$ MB. The LLC miss rate is below 0.1% for N up to and including 2^{15}, 0.56% for N $= 2^{16}$ were the memory footprint is 1.75 MB, and it increases dramatically for N $= 2^{17}$ and larger problems. Results are shown in Figure 2 and Figure 3.

Task sizes S for Black-Scholes were chosen so that task scheduling overhead has little effect on performance. S = 2048 was used for large problems, and S = *max*(N/8,16) for small problems. The work-first scheduling algorithm in OmpSs was used since it gave high and stable performance. Relative standard deviation (RSD) per benchmark was typically less than 3% for N $> 2^5$.

3.3 FFTW

FFTW already supports OpenMP, which allowed us to create a straightforward OmpSs port. This was done by replacing *omp parallel* for constructs with *omp task* loop bodies, and their associated implicit barriers with *omp taskwait*. A single precision out-of-place transform was performed, which

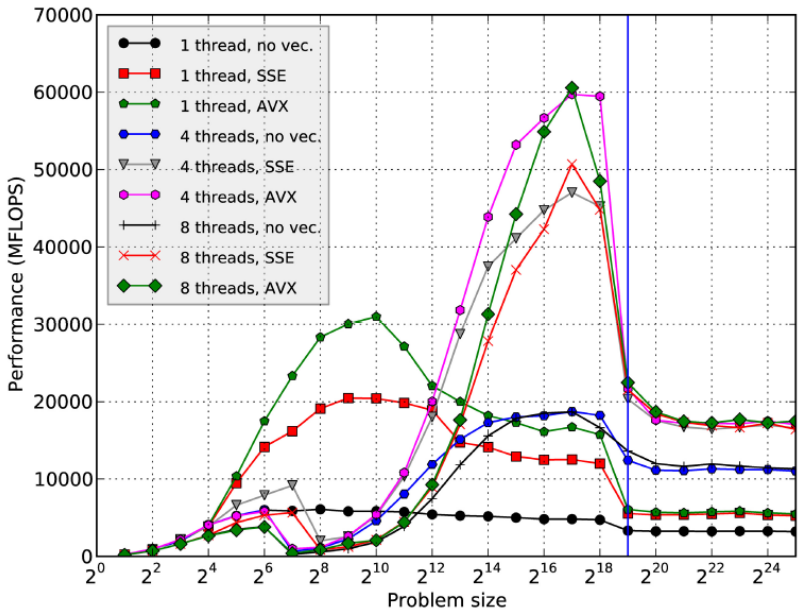


Figure 4: Performance vs. problem size for FFTW. The 8MB point is marked by the vertical line.

requires two arrays of N complex numbers each. This gives a memory footprint of 16N bytes. Thus, the largest problem that possibly could fit in the LLC is $N = 2^{19}$, as $2^{19} \cdot 16\ \text{B} = 8\ \text{MB}$. We obtained LLC miss rates less than 0.1% for problem sizes up to and including $N = 2^{17}$, and rapid increases above this limit. RSD was less than 3% for $N > 2^7$. Results are shown in Figure 4 and Figure 5.

3.4 Matrix Multiplication

Our initial experiments with the OmpSs Matrix Multiply use ATLAS with AVX. They give a peak performance at 149.7 GFLOPS running 4 threads on a 8192×8192 matrix. The peak on-chip energy efficiency is 1.97 GFLOPS/W for the same configuration, and we found the LLC misses per floating point operation to be less or equal to 3.8×10^{-5} for all problem sizes. The results are summarized in Figure 6 and Figure 7.

3.5 Discussion

We compare observations of energy efficiency improvement to corresponding parallel speedup, in order to evaluate the benefit of adding parallelism. As seen in Figures 2 and 3, Black-Scholes scales favorably. 4-thread runs

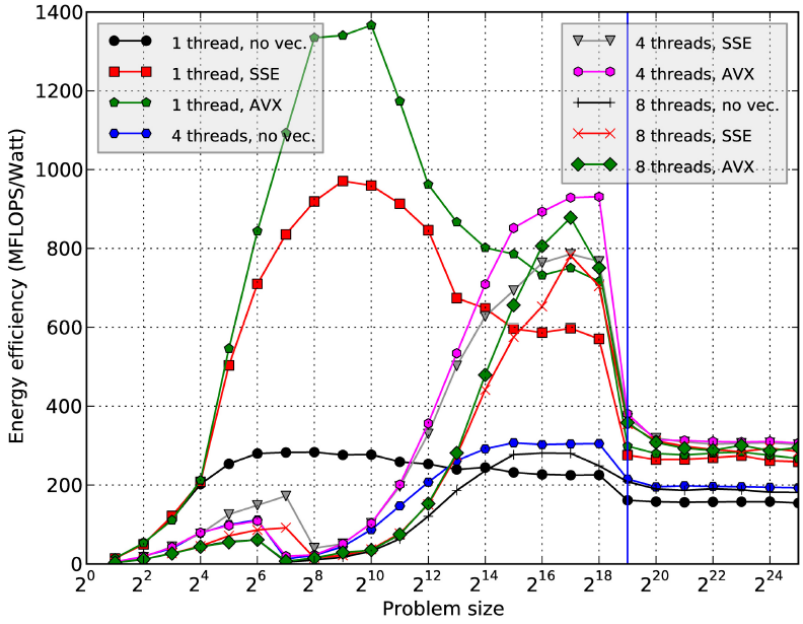


Figure 5: Energy efficiency vs. problem size for FFTW. The 8MB point is marked by the vertical line.

become advantageous at problem sizes $N = 2^{12}$ and $N = 2^{13}$, and 8-thread runs show energy benefits upwards of $N = 2^{14}$. It is also visible that Black-Scholes retains energy efficiency for out-of-cache problem sizes, albeit with a peak at $N = 2^{16}$. Speedup with hyperthreading (8 threads) is distinctly sub-linear, but there is a clear improvement which admits evaluation of the return on energy investment.

Figures 4 and 5 show that FFTW reaps no benefit from hyperthreading, and clearly becomes bandwidth bound for problem sizes beyond available cache space. This limit is characteristic of the kernel, and also witnessed by the results of Frigo and Johnson [14]. For problem sizes up to $N = 2^{14}$, energy efficiency is higher for vectorized single-thread than for parallel execution, and AVX provides further benefits over SSE. It is interesting to note that the intersection coincides with L2 cache size. For the last-level cache problem sizes of 2^{14} through 2^{18}, 4-thread execution provides higher energy efficiency, in proportion to the speedup. For matrix multiplication, Figures 6 and 7 show that even though eight threads perform significantly better than one, energy efficiency is lower for all problem sizes due to a higher energy consumption rate. As the ALU and L1/L2 caches are shared

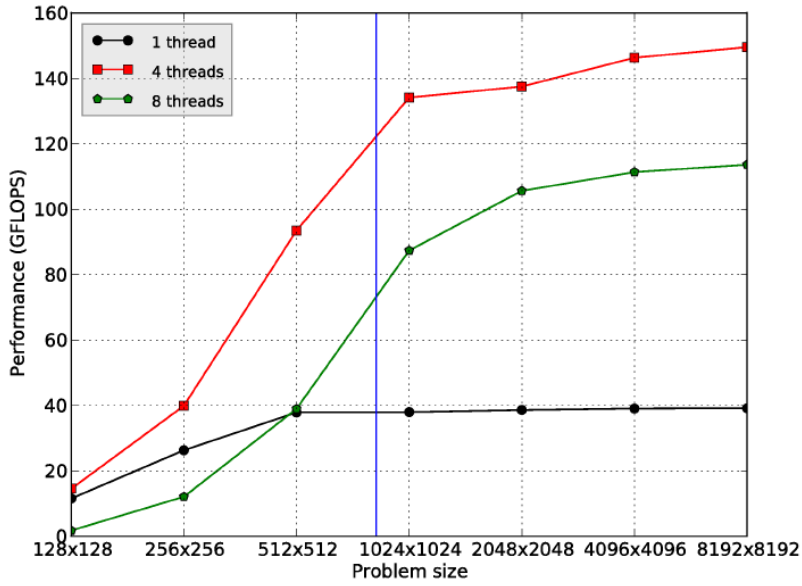


Figure 6: Performance in MFLOPS of matrix multiplication for different problem sizes. The 8MB point is marked by the vertical line.

between hyperthreads on a single core, the performance using eight threads is lower than with four, because tiled, dense matrix-matrix multiplication is computation bound.

4 Related Works

Duran et al. [4] evaluate OmpSs implementations of Black-Scholes and Matrix Multiply, but focus on performance only. Comparing with their 4-core result, we get a performance improvement in excess of factor 10. We attribute the difference to the higher CPU clock frequency of our test system, and AVX vectorization. Ge et al. [15] show how the PowerPack framework can be used to study in depth the energy efficiency of parallel applications on clusters with multi-core nodes. The framework is measurement based, and can be used to identify the energy consumption of all major system components. Li and Martinez [16] develop and use an analytical model of the power- performance implications of degree of parallelism and voltage/-frequency scaling. They confirm their analytical results by detailed simulation. Molka et al. [17] discuss weaknesses of the Green500 list with respect to ranking HPC system energy efficiency. They introduce their own benchmark using a parallel workload generator to stress main components

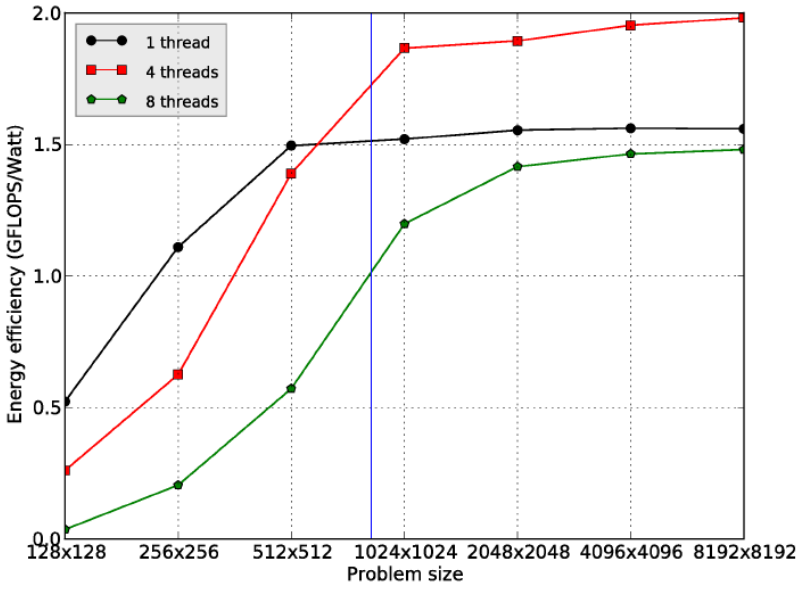


Figure 7: Energy efficiency in MFLOPS/watt of matrix multiplication for different problem sizes. The 8MB point is marked by the vertical line.

in a HPC system. Anzt et al. [18] present an energy performance analysis of different iterative solver implementations on a hybrid CPU-GPU system. The study is based on empirical measurements, and energy is saved by using DVFS (Dynamic Voltage and Frequency Scaling) to lower the CPU clock frequency while computations are offloaded to the GPU.

5 Conclusion and Future work

Using chip energy performance counters to instrument three floating-point intensive benchmarks, our experiments show that vectorization provides a significant improvement in on-chip energy efficiency, and that energy efficiency varies with problem size in common application kernels. In our results we have seen that vectorization improves both performance and energy efficiency, while the performance improvement from thread parallelism does not necessarily imply a better energy efficiency. Variation of energy efficiency with task size suggests that energy-aware task scheduling may adapt task sizes for energy efficient execution, which provides an interesting direction for future research. We also plan to extend the work by studying the impact of varying CPU clock frequencies, OmpSs scheduling policies, and using Turbo Boost Technology. We will apply the Intel Energy

Checker SDK and Yokogawa WT210 Power analyzer, to refine energy profiles by including off-chip bandwidth and memory system parameters. The experiments will be extended to a SGI Altix ICE X supercomputer, featuring 2×8 Sandy Bridge multi-core processors.

References

[1] *Mont Blanc Project Website*. URL: http://www.montblanc-project.eu/.

[2] *The Green 500 - Ranking the World's Most Energy Efficient Supercomputers*. URL: http://www.green500.org.

[3] J.M. Perez, R.M. Badia and J. Labarta. 'A Dependency-aware Task-based Programming Environment for Multi-core Architectures'. In: *International Conference on Cluster Computing*. Oct. 2008, pp. 142–151.

[4] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell and Judit Planas. 'OmpSs - A Proposal for Programming Heterogeneous Multi-core Architetcures'. In: *Parallel Processing Letters* 21 (Mar. 2011), pp. 173–193.

[5] Alex Ramirez. *European Scalable and Power Efficient HPC Platform Based on Low-Power Embedded Technology*. Presentation at the EESI conference. Oct. 2011. URL: http://www.eesi-project.eu/.

[6] R. Clint Whaley, Antoine Petitet and Jack J. Dongarra. 'Automated Empirical Optimizations of Software and the ATLAS Project'. In: *Parallel Computing* 27.12 (Jan. 2001), pp. 3–35. ISSN: 0167-8191.

[7] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Apr. 2012. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[8] Intel. *Avoiding AVX-SSE Transition Penalties*. Nov. 2011. URL: http://software.intel.com/file/39798.

[9] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, Christos Kozyrakis and Justin Meza. 'Models and Metrics to Enable Energy-Efficiency Optimizations'. In: *Computer* 40.12 (Dec. 2007), pp. 39–48.

[10] Hallgeir Lien. 'Case Studies in Multi-core Energy Efficiency of Task Based Programs (preliminary title)'. MA thesis. Norwegian University of Science and Technologoy, 2012.

[11] Intel. *Intel 64 and IA-32 Architectures Software Development Manual*. Dec. 2011. URL: http://download.intel.com/products/processor/manual/325462.pdf.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li. ' The PARSEC Benchmark Suite - Characterization and Architectural Implications'. In: *Proceedings of the Conference on International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. 2008, pp. 72–81.

[13] Stephen L. Moshier. *Cephes Math Library*. URL: http://www.netlib.org/cephes.

[14] M. Frigo and S.G. Johnson. 'The Design and Implementation of FFTW3'. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 216–231.

[15] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li and K.W. Cameron. 'PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications'. In: *Parallel and Distributed Systems, IEEE Transactions on* 21.5 (May 2010), pp. 658–671. ISSN: 1045-9219.

[16] Jian Li and José F. Martínez. 'Power-Performance Considerations of Parallel Computing on Chip Multiprocessors'. In: *ACM Transactions on Architecture and Code Optimization* 2.4 (Dec. 2005), pp. 397–422. ISSN: 1544-3566. URL: http://doi.acm.org/10.1145/1113841.1113844.

[17] Daniel Molka, Daniel Hackenberg, Robert Schone, Timo Minartz and Wolfgang Nagel. 'Flexible Workload Generation for HPC Cluster Efficiency Benchmarking'. In: *Computer Science - Research and Development* (), pp. 1–9. ISSN: 1865-2034. URL: http://dx.doi.org/10.1007/s00450-011-0194-9.

[18] Hartwig Anzt, Maribel Castillo, Juan C. Fernández, Vincent Heuveline, Francisco D. Igual, Rafael Mayo and Enrique S. Quintana-Ortí. 'Optimization of Power Consumption in the Iterative Solution of Sparse Linear Systems on Graphics Processors'. In: *Computer Science - Research and Development* 27.4 (Nov. 2012), pp. 299–307. ISSN: 1865-2042. DOI: 10.1007/s00450-011-0195-8.

Paper A.2

Performance and Energy Efficiency Analysis of Data Reuse Transformation Methodology on Multicore Processor

Abdullah Al Hasib, Per Gunnar Kjeldsberg, and Lasse Natvig

Euro-Par 2012: Parallel Processing Workshops

Abstract

Memory latency and energy efficiency are two key constraints to high performance computing systems. Data reuse transformations aim at reducing memory latency by exploiting temporal locality in data accesses. Simultaneously, modern multicore processors provide the opportunity of improving performance with reduced energy dissipation through parallelization. In this paper, we investigate to what extent data reuse transformations in combination with a parallel programming model in a multicore processor can meet the challenges of memory latency and energy efficiency constraints. As a test case, a “full-search motion estimation" kernel is run on the Intel® Core™ i7-2600 processor. Energy Delay Product (EDP) is used as a metric to compare energy efficiencies. Achieved results show that performance and energy efficiency can be improved by a factor of more than 6 and 15, respectively, by exploiting a data reuse transformation methodology and parallel programming model in a multicore system.

1 Introduction

The rapid growth of microprocessor performance for the last two decades has provided us the opportunity to solve increasingly advanced problems that require very large scale computations. However, memory latency has not been improved at a comparable rate and has become a major limiting factor for system performance. System performance is further impeded by battery capacity for handheld devices and by heat dissipation constraints for high performance processor designs [1]. Therefore, improvement of energy efficiency and memory latency has now become a major concern in contemporary computer architectures.

For data-dominated applications such as multimedia algorithms, Data Transfer and Storage Exploration (DTSE) offers a complete methodology for obtaining and evaluating a set of data reuse transformations in terms of memory energy [2]. The fundamental idea behind data reuse transformations is to move the data accesses from background memories to smaller and less energy intensive foreground memory blocks in a systematic way. This approach eventually results in significant energy savings.

In this paper, we present a technique that combines a data reuse transformation methodology with a parallel programming model to improve energy efficiency. We evaluate the performance and energy efficiency of our combined technique on a quad-core processor. We have used a “full-search motion estimation" algorithm as our test application.

This paper is organized as follows: Section 2 describes related work. Section 3 presents our methodology to improve energy efficiency and 4 illustrates our methodology using the *motion estimation* algorithm. Section 5 presents and discusses our results. Finally, we conclude the paper in Section 6.

2 Related Work

Research on data reuse transformation methodologies for multimedia applications has been actively performed for the last few decades and has led to numerous approaches to improve memory latency. Wuytack *et al.* [2] present a formalized methodology for data reuse exploration to reduce memory energy consumption by exploiting temporal locality of memory accesses using an optimized custom memory hierarchy. It is taken further and oriented towards predefined memory organizations in [3]. In [4, 5], the authors evaluated the effect of data reuse decisions on power, performance

and area in embedded processing systems. The effect of data reuse transformations on a general purpose processor has been explored in [6]. In [7], the authors presented the effect of data reuse transformations on multimedia applications on application specific processors. The research described so far emphasizes on single-core systems and relies on simulation based modeling when energy efficiency is estimated.

In [8], Kalva *et al.* presented the effect of parallel programming on multimedia applications but it lacks energy efficiency analysis. In [9], Chen *et al.* presented different optimization opportunities of the Fast Fourier Transform (FFT) to achieve a high performance implementation on IBM Cyclops-64 chip architecture. In [10], Zhang *et al.* presented an inter-core data reuse technique to exploit all the available cores to boost overall application performance. These earlier studies on parallel architectures emphasize performance rather than energy efficiency. In contrast, we have performed energy efficiency analysis on a state-of-the-art multicore processor with four cores and have used Model Specific Registers of the processor to accurately measure the consumed energy.

The previous work closest to ours is done by Marchal *et al.* [11]. It presents an approach for integrated task-scheduling and data-assignment for reducing SDRAM costs for multi-threaded applications. It does not couple the data reuse analysis with a parallel programming model the way we do here, however.

3 Energy Efficient Methodology for Multicore Processor

In this paper, we present an approach that combines the concepts from a *data reuse transformation* methodology with a *parallel programming* model to get better performance and energy efficiency.

Data Reuse Transformation: The fundamental concept of a data reuse transformation is to optimize an application and/or introduce a custom memory hierarchy to exploit the temporal locality of data accesses [2]. The memory hierarchy consists of layers of gradually smaller memories. The application code is optimized so that data that is accessed multiple times, i.e., has a high reuse factor, is copied from larger to smaller memories closer to the data path. Unless the memory hierarchy is fixed, the size and interconnect of each layer can also be optimized. For data-intensive applications, this approach gives significant energy savings since smaller memories consume less energy per access.

Parallel Programming: Multicore processors can achieve higher perform-

ance with lower energy consumption compared to a uniprocessor system. It is, however, a challenging job to develop efficient parallel applications that exploit the advantages of hardware parallelism. Different parallel programming models have been developed that can speed up applications when multiple threads or multiple processes are used [12]. At this level, parallel programs can be written using multi threaded programming and using explicit threading supported by the operating system or using programming frameworks such as OpenMP [13].

In our approach, initially we have applied different possible data reuse transformations described in [2] to optimize energy efficiency for a given algorithm. The first step identifies data sets that are reused multiple times within a short period of time, i.e., *copy candidates*. For each of the identified data sets, a copy to a smaller memory can be introduced so that data is accessed using less energy. Based on a cost trade off with extra copying of data and chip area overhead, a hierarchical memory organization is generated and an optimized set of *copy candidates* are utilized. After data reuse optimization, we develop a parallel algorithm based on the optimized solution.

Algorithm 1 Sequential Unoptimized Motion Estimation Algorithm [14]

```
1: for g=0; g<H/n; g++ do
2:     for h=0; h<W/n; h++ do
3:         Δopt[g][h] = +∞
4:         for i=-m; i<m; i++ do
5:             for j=-m; j<m; j++ do
6:                 Δ = 0
7:                 for k=0; k<n; k++ do
8:                     for l=0; l<n; l++ do
9:                         Δ+= abs(Cur[g×n+k][h×n+l] - Ref[g×n+i+k][h×n+j+l])
10:                    end for
11:                end for
12:                Δopt[g][h] = min(Δ, Δopt[g][h])
13:            end for
14:        end for
15:    end for
16: end for
```

4 Demonstrator Application: Motion Estimation Kernel

We have used a “full-search motion estimation" algorithm to evaluate the performance and energy efficiency of our combined approach.

4.1 Sequential Unoptimized Motion Estimation Algorithm

Motion Estimation (ME) is a core part of different video compression algorithms. Block-based ME algorithms involve finding the candidate block within a specified search area in a *reference frame* that is most similar to the current block in the *current frame*. A “full-search motion estimation" algorithm performs an exhaustive search over the entire search region to find the optimal solution. This process is computationally intensive and costs about 80% of the encoding time [8]. Therefore, we have chosen it as a test application in our experiment.

Full-search motion estimation is illustrated in Algorithm 1. The implementation of the ME algorithm consists of a number of nested loops. The basic operation at the innermost loop consists of an accumulation of pixel differences, while the basic operation two levels higher in the loop hierarchy consists of the selection of the new minimum. This algorithm is a sequential implementation without exploiting any data reuse transformation technique and referred as *sequential unoptimized* solution in this paper. For our experiment, we have used parameters of the QCIF format *(W=176, H=144, m=n=8)* [14].

4.2 ME Optimization Using Data Reuse Transformations

We have followed a systematic approach presented in [2] to transform the Basic ME Algorithm into an optimized solution that maps selected copies of data on a memory hierarchy to exploit temporal locality. Fig. 1 presents different possible transformations for the ME algorithm.

Each branch in the *copy candidate* tree corresponds to a potential memory hierarchy for different data-reuse transformations. Dashed lines in the figure indicate levels of the hierarchy. Each rectangle in the hierarchy corresponds to a *copy candidate*, i.e., a block of data that can benefit from being accessed multiple times from the given hierarchy level. Each *copy candidate* is annotated with its size. The highlighted path in the figure indicates a *3 layer memory* hierarchy for data reuse transformations on the *reference frame*. The hierarchy is comprised of a $H \times W$ block for the full frame, a $(2m+n-1) \times (2m+n-1)$ block and a $(2m+n-1) \times n$ block for smaller *copy candidates*. In addition, a *2 layer memory* hierarchy for the *current frame* with

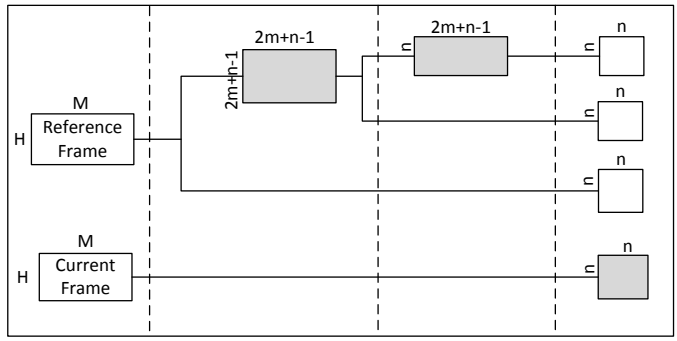


Figure 1: Copy candidate tree for data reuse decision for Motion Estimation Algorithm. The process of constructing such copy candidate tree is explained in[2]

a $H \times W$ frame memory and a $n \times n$ *copy candidate* is also introduced.

To evaluate the performance and energy efficiency of the different data reuse transformations presented in Fig. 1, the basic ME algorithm has been modified into different versions to exploit different possible transformations. Achieved performance and energy efficiency of all transformed algorithms are then measured and compared. The transformation that provides the best energy efficiency is converted to a parallel program. Algorithm 2 depicts an example of the transformed ME algorithm with two layers. The transformed algorithm introduces a smaller memory block *(Buffer)* to which the *copy candidate* is copied.

4.3 Parallel Optimized Motion Estimation Algorithm

The Motion Estimation algorithm also exhibits important properties of data parallelism. In QCIF format, a video frame is comprised of a fixed number of macro blocks (8×8 non-overlapping blocks). Prediction for a given block is determined by finding a block in a given search range of the *reference frame* that is closest to the current block. For each macro block (MB), this estimation can be done in parallel. Algorithm 2 represents our parallel ME algorithm. We have made our *2 layer* ME algorithm parallel by adding the *#pragma omp parallel for* directive of the OpenMP programming model [13] at the outermost *for* loop. This directive will instruct the compiler to distribute the work done in the for-loop immediately following the directive among all processors (cores) of the system. Variables *Ref, Cur* and Δ_{opt} are shared among the threads whereas *(h, Buffer)* are private to each thread. Note that threads should be properly synchronized while

Algorithm 2 Parallel Optimized Motion Estimation Algorithm

```
1: #pragma omp parallel for shared(Ref, Cur, Δopt) private(h, Buffer)
2: for g=0; g<H/n; g++ do
3:     for h=0; h<W/n; h++ do
4:         for k=0; k<2m+n-1; k++ do
5:             for l=0; l<2m+n-1; l++ do
6:                 Buffer[k][l]=Ref[g×n-m+k][h×n-m+l]
7:             end for
8:         end for
9:         Δopt[g][h] = +∞
10:        for i=0; i<2m-1; i++ do
11:            for j=0; j<2m-1; j++ do
12:                Δ = 0
13:                for k=0; k<n; k++ do
14:                    for l=0; l<n; l++ do
15:                        Δ+=abs(Cur[g×n+k][h×n+l]-Buffer[i+k][j+l])
16:                    end for
17:                end for
18:                #pragma omp critical
19:                Δopt[g][h] = min(Δ, Δopt[g][h])
20:            end for
21:        end for
22:    end for
23: end for
```

computing the minimum Δ_{opt}. Therefore a *#pragma omp critical* directive is used to ensure that Δ_{opt} is accessed by a single thread at a time. We have also set the *GOMP_CPU_AFFINITY* environment variable to bind each thread, i.e., each instance of the *for-loop*, to a specific core.

4.4 System Architecture and Energy Measurement

System Architecture: In our experiment, we have used the Intel® Core™ i7-2600 processor which consists of four physical cores. It supports Hyper-Threading allowing it to simultaneously process up to 8 threads, i.e., 2 threads per core. The memory hierarchy consists of a 32 KB Level-1 cache, a 256 KB Level-2 cache and an 8192 KB Level-3 cache. Level-1 and Level-2 caches are private to each core while the Level-3 cache is shared among the cores. Note that this is a memory hierarchy with a fixed number of levels and sizes, typical for a standard processor. This is different from the assumption in [2], where an application specific memory hierarchy is assumed. The base clock speed of the processor is 3.4 GHz, but it can go as high as 3.8 GHz when Turbo Boost is enabled [15].

Energy Measurement Policy: We read the non-architectural Model Specific Registers of the processor to estimate on-chip energy consumption [15]. The *MSR_PP0_ENERGY_STATUS* register gives us aggregate energy consumed by the cores as well as caches. We read this register at a fixed core frequency (3.4 GHz) and process the raw data to compute energy efficiency.

Energy Efficiency Metric: We report energy efficiency in terms of the Energy-Delay-Product (EDP) metric [16]. Measured units for Energy and Delay are *Joule(J)* and *second(s)* respectively. Therefore, the unit of the EDP metric is *Js*. Generally, the lower the EDP, the better the energy efficiency is.

OS and Compiler Parameters: We execute our experiment on OpenSuse 11.4 (x86 64) running Linux kernel 2.6.37.6. The parallel application is compiled using the *gcc* compiler with *-fopenmp* flag and optimized with *-O3* flag.

5 Results and Discussion

Energy Efficiency Evaluation of Sequential ME Algorithm:

Different data reuse transformations of the sequential ME algorithm and their corresponding energy efficiencies are presented in Fig. 2.

Fig. 2 shows that energy efficiency is improved significantly due to data reuse transformation techniques despite of the fact that such transforma-

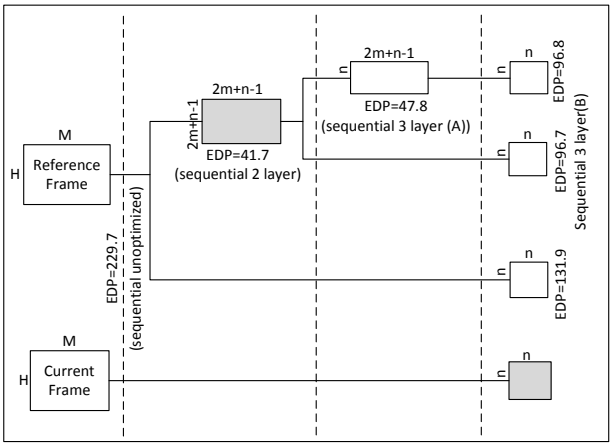


Figure 2: EDP (Js$\times 10^{-8}$) of different transformations

tions introduce both area and computational overheads. For instance, the *sequential 2 layer* transformation introduces a *(2m+n-1)×(2m+n-1)* block buffer for the *reference frame* and a *(n×n)* block buffer for the *current frame* and these additional buffers cost 2372 Bytes of area overhead. The computational and energy overhead to copy the *copy candidates* into the buffer are 0.31 microsecond and 6.67 millijoule, respectively. Therefore, in terms of EDP, the overhead is approximately 0.207×10^{-8} Js for each *new frame*. Despite these overheads, we have observed that achieved EDP for the complete handling of one *new frame* is 229.7×10^{-8} Js for the *sequential unoptimized* ME Algorithm whereas the EDP of the *sequential 2 layer* transformation is 41.7×10^{-8} Js. This improvement is attributed to the use of smaller buffers since a block of *(2m+n-1)×(2m+n-1)* integer-numbers corresponds to 2116 ($23\times23\times4$) bytes which is less than the Level-1 cache size in our system. As a result, the buffer can be brought into the Level-1 cache during the computation which significantly reduces the cost of expensive memory accesses and improves performance as well as energy efficiency.

An important observation from Fig. 3 is that the efficiency is at a peak with a *2 layer memory* hierarchy and it degrades with the introduction of any additional layers of smaller memory blocks. Two factors that contribute to this result are: (i) Additional memory layers also introduce additional area and computational overheads (ii) Smaller data blocks are copied into relatively larger cache-blocks due to the fixed-sized caches, which ultimately negate the advantage of using additional memory layers.

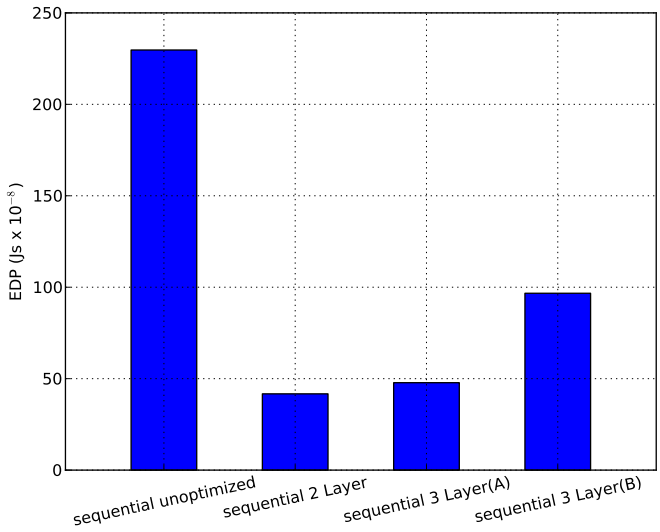


Figure 3: Data reuse transformations and their energy efficiencies measured in EDP

Energy Efficiency Evaluation of Parallel ME Algorithm:

To maximize the energy efficiency, we have converted the optimized ME algorithm that uses a *2 layer memory hierarchy* into a parallel one by using the OpenMP programming model and executed it on our system with a varying number of threads. Fig. 4 presents the obtained result.

Fig. 4 implies that parallel programming improves energy efficiency of both optimized (that exploits data reuse transformation methodology) and unoptimized solutions (not using data reuse transformations). We can see that EDP values drop rapidly with increasing number of threads and reach their minimum when 4 threads are used. Since the Intel® Core™ i7-2600 processor consists of 4 physical cores which are shared among the threads in Hyper-Threading mode, cache pollution causes the *parallel unoptimized* solution to increase the EDP values with the increasing number of threads. In contrast to the unoptimized solution, the optimized solution exhibits better cache behavior due to the use of smaller memory blocks. Hence, EDP remains almost constant during the Hyper-Threading mode.

Table 1 presents a summary of our results which reveal that data reuse transformations significantly improve energy efficiency and that the *parallel optimized* solution is the most energy efficient transformation for ME

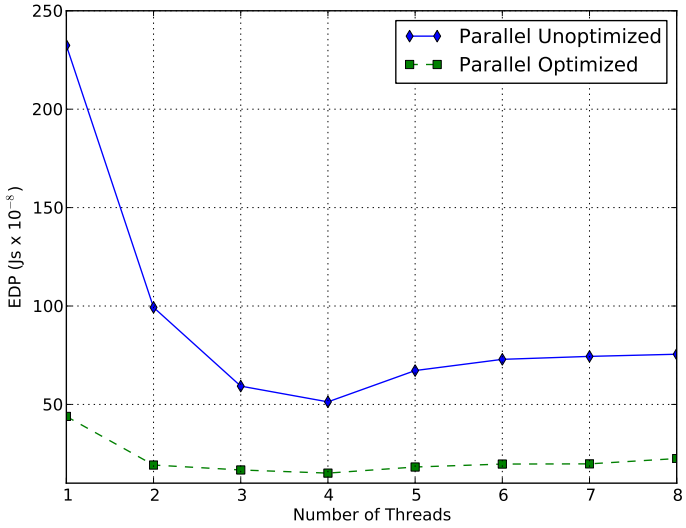


Figure 4: Improved energy efficiency using optimized parallel ME algorithm

algorithm. Normalized EDP values (with respect to *optimized parallel solution*) in the Table indicate that, *sequential optimized* and *sequential unoptimized* solutions are energy in-efficient by a factor of 2.7 and 15.1, respectively. The execution time for performing ME on one complete *new frame* is improved with a factor of 6.5 going from *sequential unoptimized* to *parallel optimized*.

In contrast to our results, in which we have obtained the best energy efficiency by using a *2 layer memory* hierarchy, Wuytack *et al.* in [2] have shown that a *3 layer memory* hierarchy is the most energy efficient scheme for the ME algorithm. However, our experiments differ from their as follows: First, we have experimented on a processor with a memory hierarchy

Table 1: Results of different data reuse transformations

Version	**Execution Time** Second$\times 10^{-6}$	**Energy** Joule$\times 10^{-3}$	**Energy Efficiency (EDP)** Js$\times 10^{-8}$	**Normalized EDP**
Sequential Unoptimized	10.9	210.7	229.7	15.1
Sequential 2 Layer	4.3	97.0	41.7	2.7
Sequential 3 Layer	4.6	104.1	47.8	3.1
Parallel Unoptimized	3.2	161.4	51.6	3.4
Parallel Optimized	1.7	89.7	15.2	1.0

of fixed sized cache-blocks. The *copy candidates* are hence mapped to a portion of these fixed-size system caches, and consequently our measurements consider the energy consumed by both used and idle cache lines. Wuytack *et al.* did their experiment in a simulation environment that created a hierarchy of memory blocks that perfectly fit the data blocks. Therefore, extra energy consumption due to unused cache area is avoided. To avoid extra energy consumption in our experiment, we would need to have an execution platform using a concept like *drowsy cache* [17] that powers down unused parts of the cache. This would give more comparable results between the two methods. It is not available in the Core™ i7 processor, however. Second, we have measured energy efficiency of the complete program rather than a part of the program that deals with data transfer. Third, we have measured on-chip memory and core energy consumption rather than considering only memory energy consumption. Fourth, their simulation environment assumes that data can be directly copied from a low-level hierarchy to a high-level hierarchy bypassing any intermediate layer. This is not possible in our system.

6 Conclusion

In this paper, we have investigated performance and energy efficiency effects of applying data-reuse transformations on a multicore processor running a motion estimation algorithm. We have shown that for a sequential *Motion Estimation* kernel, energy efficiency can be improved up to 5.5 times by using appropriate data-reuse transformation techniques, which can be further extended to 15.1 times by incorporating the OpenMP parallel programming model. We have also shown that Hyper-Threading degrades both performance and energy efficiency of the unoptimized solution. This gives clear indications that a data reuse transformation methodology in combination with a parallel programming model can significantly save energy as well as improve performance of this type of applications running on multicore processors.

References

[1] Susanne Albers. 'Energy-Efficient Algorithms'. In: *Communications of the ACM* 53.5 (May 2011), pp. 86–96. DOI: 10.1145/1735223.1735245.

[2] J.Ph. Diguet, S. Wuytack, F. Catthoor et al. 'Formalized Methodology for Data Reuse Exploration for Low-Power Hierarchical Memory

Mappings'. In: *IEEE Transactions on VLSI Systems* 6 (1998), pp. 529–537.

[3] Francky Catthoor, Koen Danckaert, Chidamber Kulkarni, Erik Brockmeyer, Per Gunnar Kjeldsberg, Tanja Van Achteren and Thierry Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002. ISBN: 9780792376897.

[4] Francky Catthoor, Sven Wuytack, G.E. de Greef and et al. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792382889.

[5] Nikos D. Zervas, Kostas Masselos and C. E. Goutis. 'Data-Reuse Exploration for Low-Power Realization of Multimedia Applications on Embedded Cores'. In: *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation*. PATMOS'99. 1999, pp. 71–80.

[6] Alexander Chatzigeorgiou, Er Chatzigeorgiou, Stamatiki Kougia and et al. *Evaluating the Effect of Data-Reuse Transformations on Processor Power Consumption*. 2001. URL: `http://egnatia.ee.auth.gr/~alec/patmos2001.pdf`.

[7] N. Vassiliadis, A. Chormoviti, N. Kavvadias and et al. 'The Effect of Data-Reuse Transformations on Multimedia Applications for Application Specific Processors'. In: *Proceedings of the International Conference on Intelligent Data Acquisition and Advanced Computing Systems Technology and Applications*. IDAACS'05. Sept. 2005, pp. 179–182.

[8] Hari Kalva, Aleksandar Colic, Adriana Garcia and Borko Furht. 'Parallel Programming for Multimedia Applications'. In: *Multimedia Tools Applications* 51.2 (2011), pp. 801–818.

[9] Long Chen, Ziang Hu, Junmin Lin and et al. 'Optimizing the Fast Fourier Transform on a Multi-core Architectures'. In: *Proceedings of the Parallel and Distributed Processing Symposium*. IPDPS'07. Mar. 2007, pp. 1–8.

[10] Yuanrui Zhang, Mahmut Kandemir and Taylan Yemliha. 'Studying Inter-core Data Reuse in Multicores'. In: *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '11. 2011, pp. 25–36.

[11] Paul Marchal, Francky Catthoor, Davide Bruni and et al. 'Integrated Task Scheduling and Data Assignment for SDRAMs in Dynamic Applications'. In: *IEEE Design & Test of Computers* 21.5 (Sept. 2004), pp. 378–387. DOI: 10.1109/MDT.2004.66.

[12] A. Podobas, M. Brorsson and K. Faxen. 'A Performance Comparison of Some Recent Task-based Parallel Programming Models'. In: *Proceedings of the International Conference on High-Performance and Embedded Architectures and Compilers*. Pisa, Italy, Jan. 2010.

[13] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. July 2011. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[14] T. Komarek and P. Pirsch. 'Array Architectures for Block Matching Algorithms'. In: *IEEE Transactions on Circuits and Systems* 36.10 (Oct. 1989), pp. 1301–1308.

[15] Intel. *Intel 64 and IA-32 Architectures Software Development Manual*. Dec. 2011. URL: http://download.intel.com/products/processor/manual/325462.pdf.

[16] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan and et al. 'Models and Metrics to Enable Energy-Efficiency Optimizations'. In: *Computer* 40.12 (Dec. 2007), pp. 39–48.

[17] Krisztián Flautner, Nam Sung Kim, Steve Martin and et al. 'Drowsy Caches: Simple Techniques for Reducing Leakage Power'. In: *Proceedings of the Annual International Symposium on Computer Architecture*. ISCA '02. Washington, DC, USA, 2002, pp. 148–157.

Paper A.3

Performance Optimization and Evaluation of a Data Cleansing Algorithm on Multicore Processors

Abdullah Al Hasib and Lasse Natvig

The 9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, 2013

Is not included due to copyright

Paper B.1

V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series

Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig

The 22nd International Conference on High Performance Computing, 2015

Abstract

Chip multiprocessors (CMPs) and heterogeneous architectures have become predominant in all market segments, from embedded to high performance computing. These architectures exacerbate on-chip data requirements, creating additional pressure on the memory subsystem. Consequently, efficient utilization of on-chip memory space becomes critical for data intensive applications. A promising means of addressing this challenge is to use an effective compression method to reduce the data transmitted along the memory hierarchy.

In this paper we present V-PFORDelta, a real-time vectorized integer differential compression method for memory bound applications. We evaluate the effectiveness of our SIMD (Single Instruction Multiple Data stream) based compression method on an industrial hydrological time series data processing kernel. We analyzed both Streaming SIMD Extensions (SSE) and Advanced Vector Extensions 2 (AVX2) versions of the compression method. Results show that the performance and energy efficiency can be improved up to a factor of 3.1 and 8.2, respectively. The proposed method not only outperforms the uncompressed SIMD implementations of the hydrological kernel, but also reduces the data storage requirements by a factor of 1.56*x* to 3.38*x*, depending on the analyzed dataset.

1 Introduction

The performance gap between processors and main memory along with the increasing data movement requirements in both memory and cache bound applications makes processors likely to spend a significant fraction of their runtime stalled [1]. Effective data compression can play an important role in reducing the amount of data to be transfered along the memory hierarchy [2]. However, the compression and decompression processes incur additional computational overheads that can nullify the benefits of data compression. Therefore, the compression algorithm has to be chosen carefully to realize the concept of improving the application performance by reducing the data transmission requirements.

Time series are widely used in many scientific and engineering fields, such as analytical studies like weekly share-prices and monthly profit analysis in economics, weather forecasting in meteorology or power generation and prediction in control engineering [3, 4]. Generally, time series data processing (TSDP) deals with fairly large data sets that exhibit good correlation among the data points. This makes TSDP an interesting domain to evaluate the effectiveness of data compression algorithms when optimizing applications for energy efficiency and performance.

Over the last few years, support for fine grained data parallelism using SIMD instructions has become prevalent in virtually every processor in the market. SIMD issue width has increased at the same rate as cores have been added to the chip, and this trend is expected to continue [5]. Some common examples of SIMD instruction set extensions include Intel's SSE and AVX family, AMD's 3DNow!, ARM NEON, Motorola's AltiVec and IBM's BlueGene/L SIMD instructions. SIMD instructions provide higher performance, better energy efficiency and greater resource utilization [6, 7]. There are many applications which can potentially benefit from SIMD instructions to improve performance. However, despite the potential of SIMD instructions in developing energy efficient applications, modern compilers still do not have adequate auto-vectorization support for complex codes [8, 9]. Therefore, when code efficiency is required, it is often written manually in assembly language or using SIMD intrinsics.

Taking these facts into consideration we present V-PFORDelta, a vectorized Patched-Frame-Of-Reference Delta compression algorithm which aims at improving the energy efficiency and performance of a cache/memory bound application while reducing the capacity requirements of the datasets. The contributions of this paper can be summarized as follows:

- We investigate several aspects of an *industrial time series application*. We believe that the overall performance of the application can be improved by exploiting the existing redundancy in the dataset, compressing data at the expense of additional computational resources during the decompression process.
- We present a vectorized differential compression method (V-PFORDelta) for cache/memory bound compute kernels. Our strategy is to increase the cache block utilization and to reduce the total number of off-chip memory accesses by using a lightweight real-time SIMD-based compression/decompression method. V-PFORDelta is based on a hybrid data structure for the compressed data (AoS[1]+SoA[2], aka AoSoA[3]), regarded best practice in SIMD programming [10]. Our proposal is also enhanced with SIMD prefetching to increase the data locality of the application.
- We evaluate the feasibility of our approach on a demonstrator application (i.e. an *industrial time series application*) with different hydrological time series datasets (both synthetic and real world). Results show that the proposed approach can provide a significant performance and energy efficiency gain over the uncompressed SIMD computations.
- We also study the effects of voltage and frequency scaling on the different implementations to further improve its energy efficiency. Frequency has a higher impact on the performance of V-PFORDelta since it requires additional computations to decompress the data, becoming more CPU-bound than the uncompressed alternatives.

2 Related Work

Integer compression has received a lot of attention over the years. However, in this literature review, we primarily focus on the lossless integer compression techniques.

Some of the earliest integer compression techniques are Delta coding [11], Variable-byte (VarByte) coding [12], Rice and Golomb coding and Elias gamma coding [13]. Delta coding encodes integer numbers by subtracting successive values. The fundamental concept is to code the first value as

[1]Array of Structures.
[2]Structure of Arrays.
[3]Array of Struct of Arrays OR Tiled Array of Structs.

it is. The remaining values are represented as the differences between successive values. A good compression ratio can be achieved by this approach if the differences between successive values are small. VarByte is a simple byte-oriented compression method. It uses a variable number of bytes to represent integer values (e.g. 34 is represented using 8 bits, 144 is represented using 16 bits). This compression technique is fast but it provides low compression ratio. Golomb and Rice coding are bit oriented coding schemes. In Golomb coding, an integer (i) is coded by quotient (q) and remainder (r) of division by the divisor (d). In case of Rice coding, the divisor used is a power of 2. These coding schemes provide good compression ratio for small integer values but the decompression speed is quite slow. In Elias Gamma coding, an integer (i) is encoded by its binary representation preceded by $\lfloor \log_2 i \rfloor$ zeros. This relatively slow technique provides good compression for small integers but not for the large ones.

In recent years, several fast compression techniques have been proposed, including the Simple family (S9, S16) [14], Frame Of Reference (FOR) or Patched Frame Of Reference (PFOR) [15]. Simple9 encoding technique packs as many integers as possible in 32 bits (one word). This encoding technique is fast and provides better compression ratio than Variable-byte. FOR and PFOR coding compress a block of numbers at a time (e.g. 128 numbers). A fixed number of bits are used to encode each regular number in case of fixed-length encoding. The numbers that cannot be encoded using fixed length bits are considered exceptions and stored using 4 bytes. In varint-G8IU [16], a variable number of integers are encoded in 8 bytes. Once encoded, they are grouped together along with a 1-byte descriptor containing the unary representations of the lengths of each encoded integer. 8 data bytes can be used to encode from 2 to 8 integers depending on the size of the encoded integers. The number of integers is encoded by the number of zeros in the descriptor. More recently, SIMD-based compression techniques have gained popularity. Stepanov et al. presented a taxonomy for variable-length integer encoding formats and presented a byte preserving integer encoding algorithm using SIMD instructions [17]. Schlegel et al. applied SIMD-based decompression algorithms to derive parallel versions of two well-known integer compression techniques: null suppression and Elias gamma encoding [16]. In [18], Ao et al. proposed a linear regression-based parallelized compression technique for lists intersection. In [19], authors proposed a parallel compression algorithm that aims to improve the instruction level parallelism by exploiting a 4-way vertical data layout format. Lemire et al. proposed a delta coding-based compression technique, FastPFOR, that uses vectorized binary packing over blocks of

128 integers [20]. This scheme stores the exceptions on a per page basis, but selects the base value *b* on a per block basis. They had shown that their approach is nearly twice as fast as the previously fastest schemes on desktop processors (varint-G8IU and PFOR). The authors have further extended their work by computing prefix sum using SIMD operations in their proposed algorithm [21].

Our compression method is primarily motivated by the algorithm presented in [21]. However, the method presented in that paper is a block-based compression method that operates on a list of sorted integers (in non-decreasing order). Whereas we propose a streaming compression method that does not assume that the data is sorted in any order. Moreover, we trade-off compression ratio in favor of decoding speed to reduce the computational overhead during the decompression process.

3 Powel Hydrological Compute Kernel

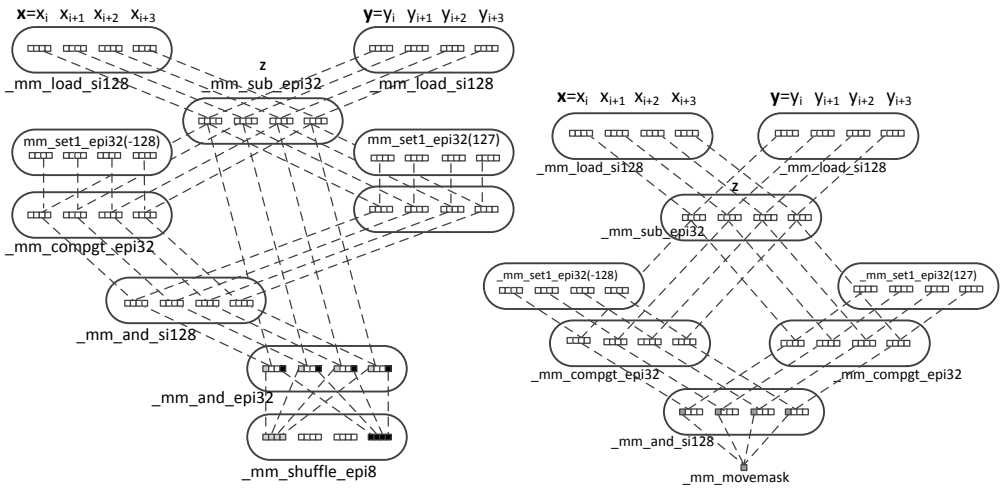
Powel AS is one of the leading providers of software and services to the utility companies in Scandinavia [22]. To evaluate the feasibility of our proposed approach, we use a hydrological time series compute kernel developed in Powel AS. The compute kernel produces a summary series from a number of time series data that receives as input from a database. Each series (s_i) in the input dataset (**ds**) can be formally defined as,

$$\mathbf{ds}=\{s_i\}_{i=0}^{n-1}, s_i = (t_i, v_i, q_i)$$

where v_i represents the hydrological in-flow measured in m^3/sec at time t_i, and q_i represents the status information (e.g. valid or invalid data) for the measurement at time t_i.

The kernel takes two time series as inputs at a time and performs computations as it passes over the data points at each time instance, generating a new time series as output. Initially, it computes time axis **t** vector of the resultant series from the given inputs. Next, for each t_i in the resultant series, the kernel takes the corresponding v_i from the input series and computes v_i of the resultant series. Any missing value in the input series is estimated using linear interpolation. The corresponding q_i parameter is also updated to indicate the value is interpolated. Finally, all the aforementioned steps are repeated until the entire dataset (**ds**) is processed.

The exact pseudo-code of this compute kernel is subject to a non-disclosure agreement (NDA), and thus can not be presented in this paper. What we can tell is that the kernel is implemented in C++, performs computationally



(a) Delta computations for 4 data points (b) 4-bit bitmask for 4 delta values

Figure 1: Data Flow Graph (DFG) of the compression algorithm. Four 32-bit integers are loaded into each 128-bit SIMD register to compute four 8-bit delta values and corresponding 4-bit bitmask. The repeating operations in the figures are only for illustration purpose; the implementation of the algorithm avoids those repeating operations.

inexpensive $O(n)$ operations, and that the data access pattern is sequential. However, the kernel needs to process a huge volume of time series data, and consequently it becomes limited by the memory subsystem. Parallelism is handled at a higher level, serving multiple requests simultaneously, further increasing the pressure on the memory hierarchy. In Section 6 we will describe different performance related aspects of this application, including performance variations when scaling core frequency, cache and memory related events and percentage of stalled CPU cycles due to lack of resources.

4 Time Series Datasets

The following hydrological time series datasets are used in our experiments.

Synthetic data (Powel dataset): We were given a real-world hydrological time series dataset by Powel. The dataset contains 5 different time series and requires 2.3 MB of internal storage. However, this dataset is not large enough to be used in our experiment since it fits on the last level cache (LLC) of the processor. We generate a synthetic dataset of 105 MB containing 250 time series as a viable alternative to a larger dataset based on the

Table 1: Required C/C++ Intrinsics to Implement V-PFORDelta Compression Algorithm

Operation	SSE	AVX2	KNC
bitwise OR	_mm_or_si128	_mm256_or_si256	_mm512_or_si512
bitwise AND	_mm_and_si128	_mm256_and_si256	_mm512_and_si512
shift right by a number of bytes	_mm_srli_si128	_mm256_srli_si256	_mm512_srli_si512
shift left by a number of bytes	_mm_slli_si128	_mm256_slli_si256	_mm512_slli_si512
add four 32-bit integers	_mm_add_epi32	_mm256_add_epi32	_mm512_add_epi32
shuffle four 32-bit integers	_mm_shuffle_epi32	_mm256_shuffle_epi32	_mm512_shuffle_epi32
compare four 32-bit integers	_mm_cmpgt_epi32	_mm256_cmpgt_epi32	_mm512_cmpgt_epi32
store a 128-bit register	_mm_storeu_si128	_mm256_storeu_si256	_mm512_storeu_si512
load to 128-bit register	_mm_loadu_si128	_mm256_loadu_si256	_mm512_loadu_si512
mask from most significant bits of 32-bit elements	_mm_movemask_ps	_mm256_movemask_ps	-
unpack and interleave 32-bit integers	_mm_unpacklo_epi32	_mm256_unpacklo_epi32	-
convert 8-bit integers to 32-bit integers	_mm_cvtepi8_epi32	_mm256_cvtepi8_epi32	-

given fractional dataset. Each series in the synthetic dataset is generated by randomly choosing a fixed number of unique v_i values in the corresponding real-world series, and updating all the occurrences of the selected values by $\pm 10\%$. Finally, all the synthetic series generated from the same original series are interleaved with the synthetic series. The data compression ratio of this dataset is 1.56, which is close to the compression ratio of the original dataset ($< 3\%$).

Real-world data (MOPEX dataset): This dataset is obtained from the National Weather Service Hydrology Laboratory which is used in Model Parameter Estimation Experiment (MOPEX) [23]. The dataset contains the records of hourly precipitation flows in *millimeter* from 438 MOPEX basins from the year 1948 to 2003. However, in our evaluation, we use the data from 20 basins out of 438 to produce a real-world dataset of 153 MB, which is large enough to be used in our experiment (i.e., does not fit on the LLC). The data compression ratio of this dataset is 3.38.

5 V-PFORDelta Compression Algorithm

TSDP datasets tend to vary smoothly over time, and hence can be encoded using delta coding effectively [20]. V-PFORDelta exploits the existing correlation among the data points by using SIMD-based differential compression and binary packing methods. SIMD operations will improve encoding and decoding speed significantly with the cost of compression ratio. We use a hybrid data structure to represent the compressed data, improving data locality. In addition, we use SIMD prefetch instructions for better memory management. The implementation details of V-PFORDelta are given for SSE. Support for AVX2 is achieved using equivalent AVX2 intrinsics and 256-bit registers (Table 1).

5.1 Encoding

V-PFORDelta coding scheme is illustrated in Algorithm 3.

Delta computation: Assume that v_1, v_2 ... v_n are the in-flow measurements to be compressed using the differential encoding technique. To enable SIMD computation, each value at index i is subtracted from the value at index $i-4$ i.e. $\delta_i = v_i - v_{i-4}$. We tune the bit width (w) of delta to 8 to minimize the value of $\{n \times b + c(w) \times 32\}$ where n is the length of time series and $c(w)$ is the number of exceptions. The process of generating 8-bit delta values using SSE intrinsics is illustrated in Fig. 1a. Each round box in the figure denotes a 128-bit vector register and each small square in the elliptical circle denotes an 8-bit value. The operations performed on the vector registers (with mm_ prefix) are denoted by SSE intrinsics with *_mm_* prefix. The dashed-lines represent the vector registers involved in a specific operation. In Fig. 1a, two vectors $\mathbf{x}=\{v_{i+1}, v_{i+2}, v_{i+3}, v_{i+4}\}$ and $\mathbf{y}=\{v_{i-3}, v_{i-2}, v_{i-1}, v_i\}$ are used to compute four 8-bit delta values. Each element of the vectors is shown separately as x_i, y_i. First, elements x_i and y_i are loaded into the SSE registers (*_mm_load_si128*). Then y_i is subtracted from x_i (*_mm_sub_epi32*) and the result z_i is checked (*_mm_cmpgt_epi32*) to be within the range [-128, 127] or not (range is set using *_mm_set_epi32*). If z_i is not within the range, it will be set to 0 using *_mm_and_si128*. The low-order 8-bits of each z_i in the vector register are placed in consecutive locations (*_mm_shuffle_epi8*).

Binary packing: Generally, in-flow measurements tend to vary smoothly over time, and it is expected that a significant portion of the delta values can be represented using a small number of bits. We tweak this compression parameter allocating 8-bits for each delta value (δ_i). Values that do not fall into this limit (i.e. $\delta_i < -128$ or $\delta_i > 127$) are considered *exceptions*. When an *exception* is detected, the corresponding delta value (δ_i) is set to 0 and a 32-bit field is used to hold the original value (v_i). Similar to the delta sequence, *exceptions* are stored in consecutive memory locations, but in a separate location from the delta sequence. To distinguish between the *exceptions* and the regular delta values, a 1-bit mask (m_i) is used for each δ_i. The bitmask sequence is placed right after the delta sequence in the memory. The process of generating a 4-bit bitmask using SSE intrinsics is illustrated in Fig. 1b. In the first four levels of the figure, data values are loaded into 128-bit vector registers, delta values are computed and then checked to be within the range [-128, 127]. Finally, the most significant bits of each 32-bit resultant values are extracted to generate the 4-bit bitmask (*_mm_movemask_ps*).

Algorithm 3 V-PFORDelta Encoding Algorithm
Input: a list of 32-bit integers (X), shuffle table
Assume: length(X) is multiple of 4
Output: vectors of delta (δ), bitmask (m) and exceptions (exp)

```
 1: mm_x ← load(X[0])                          ▷ load 4 integers into SIMD register
 2: i ← 0                                      ▷ loop counter
 3: while i < length(X) do
 4:     mm_y ← load(X[i])                      ▷ load next 4 integers
    Phase 1 - Delta computation
 5:     mm_z ← mm_x - mm_y                     ▷ compute 32-bit delta
 6:     δ ← shuffle(mm_z, MASK)                ▷ generate 8-bit delta
 7:     δ ← shiftLeft(δ, 1)
    Phase 2 - Bitmask computation
 8:     mm_lowerBound ← mm_z > -128            ▷ z_i > -128 ?
 9:     mm_upperBound ← mm_z < 128             ▷ z_i < 128 ?
10:     mm_range ← mm_lowerBound & mm_upperBound
11:     bitmask ← moveMask(mm_range)           ▷ extract MSB
    Phase 3 - Exception compaction
12:     m ← lookup(shuffleTable, bitmask)      ▷ extract shufflemask m
13:     exp ← shuffle(mm_y, m)                 ▷ compact exception stream exp
14:     mm_x ← mm_y                            ▷ update X-vector
15:     i ← i+4                                ▷ update loop counter
16: end while
```

Algorithm 4 V-PFORDelta Decoding Algorithm
Input: baseT, baseV, baseQ, comp, bitmask, exp
Output: time, value, status

```
 1: loadData(baseT, baseQ, exp, comp.deltaT, comp.deltaQ, comp.deltaV)
 2: for i=0 to 15 do
 3:     time[i] ← expandAndAdd(mm_deltaT, mm_baseT)
 4:     status[i] ← expandAndAdd(mm_deltaQ, mm_baseQ)
 5:     mm_baseV ← getBase(mm_exception, bitmask)
 6:     value[i] ← expandAndAdd(mm_deltaV, mm_baseV)
 7:     mm_deltaT ← shiftRight(mm_deltaT, bitmask, 4)
 8:     mm_deltaQ ← shiftRight(mm_deltaQ, bitmask, 4)
 9:     mm_deltaV ← shiftRight(mm_deltaV, bitmask, 4)
10:     i ← i+4
11: end for
```

Exception compaction: Exceptions are stored in consecutive memory locations, but the order of appearance in the data sequence is random. There-

fore, we need to compact the exception stream to place the sparsely distributed exceptions one after another. This is done by identifying the values v_i in the data stream that cause the exceptions using a *bitmask*, and then reordering the v_i to put all the exceptions together. Relative positions of exceptions in a vector register are defined by a *shuffle-mask*, which is generated from the *bitmask* sequence. The *shuffle-mask* generation process is computationally expensive. To reduce the associated computational overhead, these masks (i.e. 16 *shuffle-masks* for both SSE and AVX2 based computations) are precomputed and stored in a *shuffle-table*.

Let us explain the exception stream compaction process with a simple example. Assume that y_i = {10234, 2010, 5323, 3050} and x_i = {10000, 2000, 5000, 3000} for i=1...4; So the delta, $\delta_i = y_i$-x_i = {234, 10, 323, 50} and the corresponding bitmask, m_i = 0101. Consequently, the shuffle-mask is calculated as, s_i = {1, 3, 128, 128}. After the stream compaction phase, the final result will be: {10234, 5323, 0, 0}. SIMD DFG of the exception compaction is depicted in Fig. 2.

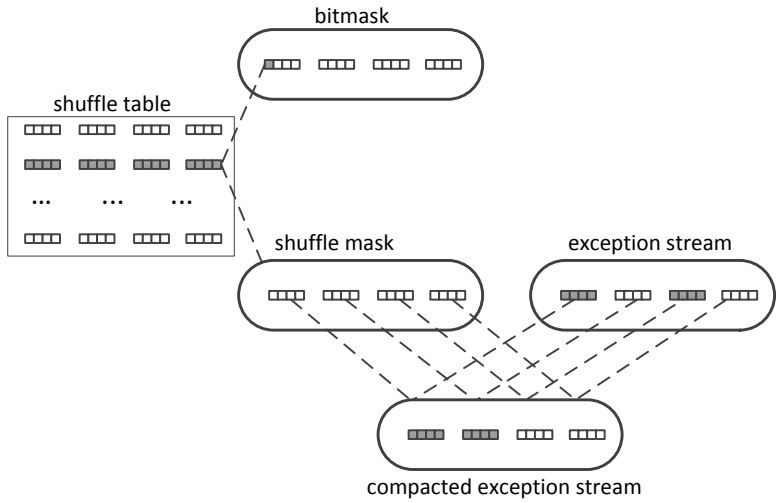


Figure 2: Exception stream compaction using SSE permutation. Bitmask identifies the exceptions and points to a specific shuffle mask to permute exceptions and place them sequentially in memory.

5.2 Decoding

The decoding algorithm takes the initial time (*base_T*), status values (*base_Q*), compressed data (*comp*), and stream of exceptions (*exp*) as input, and decompresses a block of 16 data elements at a time. Algorithm 4 presents

our proposed SIMD-based decoding algorithm. In the decoding process, the first 8-bit delta values are loaded into a SIMD register and extended into 32-bit signed integers values using *_mm_cvtepi8_epi32*. Similarly, the exception streams are loaded into 4 different SIMD registers. After that, the first 32-bits in each SIMD registers are unpacked and interleaved using *_mm_unpacklo_epi32*. This way, we generate the base values of the differential codes which are stored in a separate SIMD register. These base values are then added with the 32-bit delta values in order to produce the original values. Finally, based on the bitmask value, the low order 32-bits of each exception stream are shifted out of the registers (*_mm_srli_epi32*). The SIMD DFG of the decoding process is illustrated in Fig. 3.

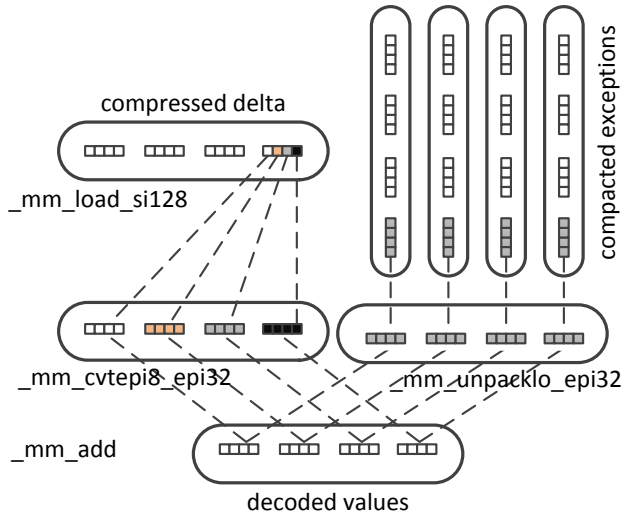


Figure 3: DFG of the decoding process. 8-bit delta values are expanded to 32-bit values and added with the base-values to generate 32-bit integers.

5.3 Compressed data structure and prefetching

We introduce the following data structure for the compressed time series dataset in order to improve the data locality property as well as to facilitate vector computations.

```
struct CTSstruct {
    int16_t delta_t[16];
    int8_t delta_v[16];
    int8_t delta_q[16];
 };
```

Table 2: Hardware Specifications of the Test Platforms

Processor	**Intel® Core™ i7-2600**	**Intel® Core™ i7-4700K**
Architecture	Sandy Bridge	Haswell
Clock Speed	1.6 – 3.4 GHz	0.8 – 3.5 GHz
# of Cores	4 physical cores with 8 threads	
L1 Cache	32 KB data + 32 KB instruction, private, 8-way associativity	
L2 Cache	256 KB, private, 8-way associativity	
L3 Cache	8 MB, shared, 16-way associativity	

We keep the size of *CTSstruct* comparable to the cache line size of our test platforms (i.e. 64 Bytes in Sandy Bridge and Haswell system), and pack several compressed values of time, status and in-flow measurements. This hybrid between array of structures + structure of arrays keeps data in the same cache line to improve the spatial locality of the compressed data structure. A typical computation in the Powel compute kernel involves one time value, one status value and one inflow-status value from each source and destination series. The designed data structure contains all the three required elements in the same cache line so that the elements can be loaded from the memory into the cache using a single memory access. Additionally in *CTSstruct*, a group of 16 time values are compressed and stored in consecutive memory locations. This is true for *status* and *in-flow* data as well. Consequently, *CTSstruct* facilitates SIMD-based computations as all the 16 `delta_v` or `delta_q` elements or 8 `delta_t` elements can be loaded into a 128-bit vector register using a single operation (*_mm_load_si128*). We further optimize the performance of V-PFORDelta by manually inserting prefetch requests in the code. To enable software prefetching, we use SIMD prefetch intrinsics (*_mm_prefetch*) [24] and set the cache-level placement hint to *_MM_HINT_T0* so that each block of data in the *CTSstructure* is brought into L1 cache in advance. This will eventually reduce memory latency provided that the prefetch requests are sent early enough.

6 Results and Discussion

This section presents the results of a set of experiments we conduct to demonstrate the effectiveness of our proposed approach. For this purpose, we implement the following five variants of Powel kernel:

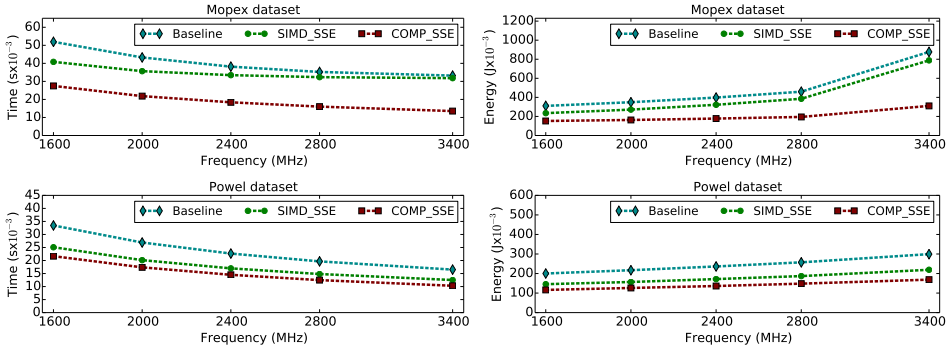
- *Baseline*: Powel kernel implemented using C++ programming language.

- *SIMD_SSE*: Vectorized Powel kernel using SSE instructions.
- *SIMD_AVX2*: Vectorized Powel kernel using AVX2 instructions.
- *COMP_SSE*: SSE-vectorized Powel kernel with integrated V-PFORDelta coding.
- *COMP_AVX2*: AVX2-vectorized Powel kernel with integrated V-PFORDelta coding.

The performance and energy efficiency evaluation will be complemented by a detailed analysis of the processor's internal performance counters. This analysis includes processor stalls and cache accesses/misses for single threaded kernels. Our goal is to gain an insight of the effects of different implementations on the processor's usage of internal resources. This analysis may reveal bottlenecks to be solved in future work.

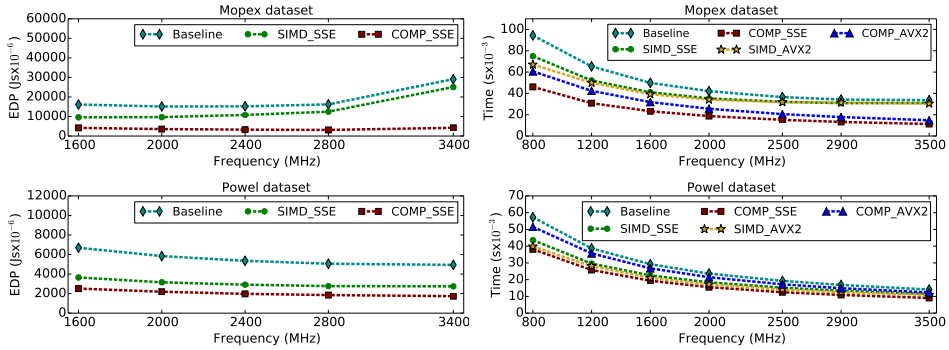
Core frequency has a linear impact on performance, as long as we have a high activity factor. However, for memory-bound applications, reducing frequency has minimal effect on performance, improving energy efficiency. As a result, the best energy efficiency for TSDP can be expected at low frequencies. Nonetheless, V-PFORDelta requires additional computations to decompress the data, making the kernel more CPU-bound. We perform a voltage and frequency analysis to detect negative/positive effects of frequency scaling on the overall performance and energy efficiency of the different implementations, and we see it reflected in our results.

We use two desktop processing systems namely Sandy Bridge (SB) and Haswell (HL) in our experiments. The hardware specifications of these systems are presented in Table 2. Both systems run with Ubuntu 14.04.1 LTS 64-bit OS. Intel C++ compiler (14.0.1) with -O3 optimization flag is used to generate the executables. Turbo Boost Technology is disabled in the BIOS and CPU frequency is set to a certain value (using *cpufreq-set*) while taking the measurements on both systems. AVX2-based kernels are evaluated only on the HL platform as the SB processor does not support AVX2. We could not extend our experiments on the Intel® many-core platforms, as the current Xeon Phi Knights Corner instruction set does not offer the full support for integer operations, which is required to implement our algorithm (see Table 1). However, we expect it to be possible in the upcoming Knights Landing co-processor, since it offers full support for AVX512 [25].



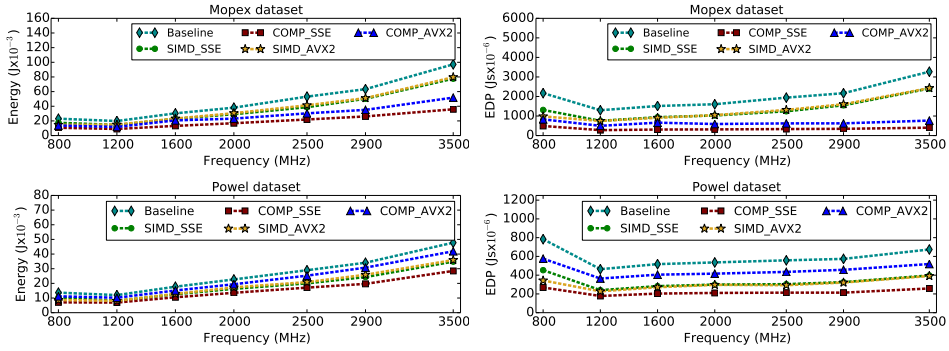
(a) Execution time on SB

(b) Energy consumption on SB



(c) Energy efficiency on SB

(d) Execution time on HL



(e) Energy consumption on HL

(f) Energy efficiency on HL

Figure 4: Execution time, core-energy consumption and energy efficiency in terms of energy delay product (EDP) at different frequencies on SB and HL systems.

6.1 Metrics used for the analysis

We use execution time (in seconds) as the metric for performance evaluation. Since the V-PFORDelta performs offline data compression, only decompression time is included in the *COMP* kernels evaluation. Energy efficiency is measured in terms of Energy Delay Product (EDP: *Js*) [26]. Generally, the lower the EDP, the better the energy efficiency. Speedup and Relative EDP at a certain frequency are computed with respect to the execution time and EDP of the *Baseline* kernel at the corresponding frequency. Both core-energy and system-energy consumptions are considered to analyze energy efficiency. We use Yokogawa WT210 external power meter to measure system-energy consumption, and read Model Specific Registers of the Intel® processors to estimate core-energy *(PP0)* consumption [24]. PAPI native counters [27] are used to track cache and memory related events, as well as DineroIV [28] (a trace-driven cache simulator) is used to analyze the nature of cache misses (i.e., compulsory, capacity and conflict). The input traces for the DineroIV are generated using the Lackey tool of Valgrind [29].

6.2 Performance analysis

In our first set of experiments, as depicted in Fig. 4, we compare performance and core-energy consumptions of different kernels on our evaluation platforms. It can be seen from the figures (Fig. 4a and 4d) that at high core frequencies performance of the *Baseline* kernel tends to be blocked by the memory subsystem, since the CPU spends many cycles waiting for data transfers (as shown in the Fig. 11). The uncompressed SIMD implementations further increase the pressure on the memory subsystem, since each operation now requires four to eight (SSE, AVX2) data values to be moved into the registers. *COMP* kernels use a portion of this idle CPU time to extract the compressed dataset. As a result, the overall performance of the TSDP does not deteriorate due to the additional computational overhead of data decompression; rather it improves as the compression helps to reduce the amount of data to be transmitted across the memory hierarchy (and thus the total waiting time). In fact, total number of L1 data cache accesses (and misses) for *COMP_SSE* kernel is reduced to half of those required by the *SIMD_SSE* kernel on SB system as shown in Table 3. It is important to note that the *COMP* kernels benefit more from increased core frequency due to the increase in computation requirements during decompression.

At a low core frequency, the *uncompressed SIMD*-based computations reduce the execution time up to 30% over the *Baseline* kernel execution time. At such low frequencies, the ratio of data that can be served by the memory

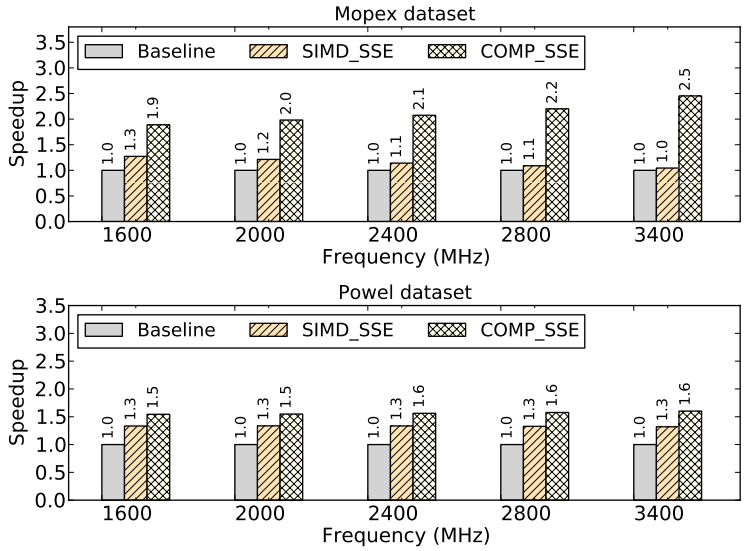


Figure 5: Speedup on SB.

subsystem increases, since frequency scaling only affects the CPU and caches. In this scenario, using a SIMD implementation provides greater benefits than at higher frequencies (specially in terms of energy and EDP), since we compute additional data per instruction. An important concluding remark is that, when the memory subsystem limits the performance benefits of a SIMD implementation, reducing the operating frequency can greatly improve the energy efficiency of the application.

We further investigate the effect of data compression ratio on the performance of *COMP* kernels. Ideally, a single SSE load contains 16 8-bit delta values that can be extracted into 16 integer values. But in reality, there exists *exceptions* in the data stream, and each *exception* contributes one additional load or shift operation. Consequently, the speedup depends on the correlation among the data points. The higher the correlation among the time series data points, the higher its compression ratio and the better its speedup. Since MOPEX dataset has better compression ratio than that of Powel, *COMP* kernels provide greater speedups (up to 3.1) for MOPEX dataset compared to Powel dataset (up to 1.8) in Fig. 5 and Fig. 6. Still, *COMP* kernels manage to outperform both the *Baseline* and the *uncompressed SIMD* kernels for all core frequencies and input sets.

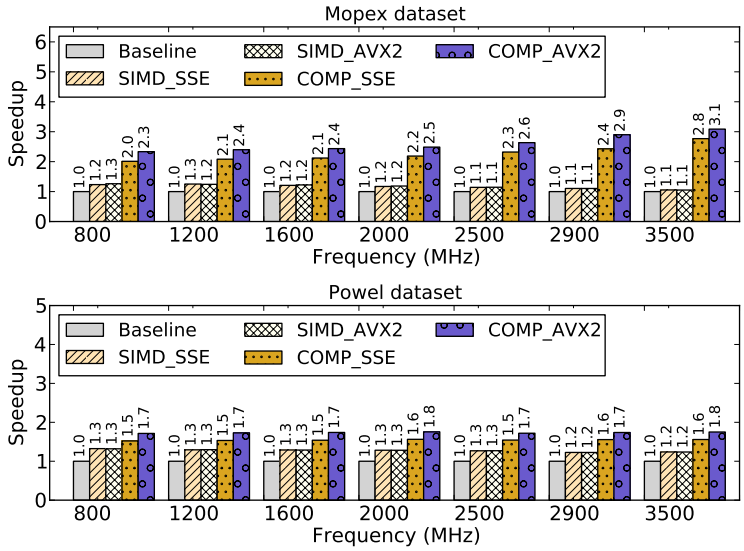


Figure 6: Speedup on HL.

6.3 Energy efficiency analysis

To analyze the energy efficiency of our proposed approach, we consider both core and system energy consumption. From the figures in Fig. 4b and Fig. 4e, we can see the expected correlation between core energy consumption and core frequency (i.e. the more frequency, the more energy). What is important to notice is that, for the *Baseline* and *uncompressed SIMD* kernels, the additional power dissipation at high frequencies does not translate into a linear performance improvement. This is, as mentioned previously, due to the *memory-bound* nature of the kernel. Computations are performed faster and cost more power, but the processor still has to wait for the memory to serve all the data. As a result, total core energy consumption increases significantly at high core frequencies. On the other hand, the *COMP* kernels consume less energy, even at high core frequencies. Our compression method reduces both on-chip and off-chip memory bandwidth requirements at the expense of additional computational requirements (decompression). As a result, the system benefits from increasing core frequency, since computations are now critical to decompress the required data. The complete (at-the-wall) system energy consumption shows similar trends in Figure 8. These measurements consider both core and uncore components (e.g. motherboard, memory, etc).

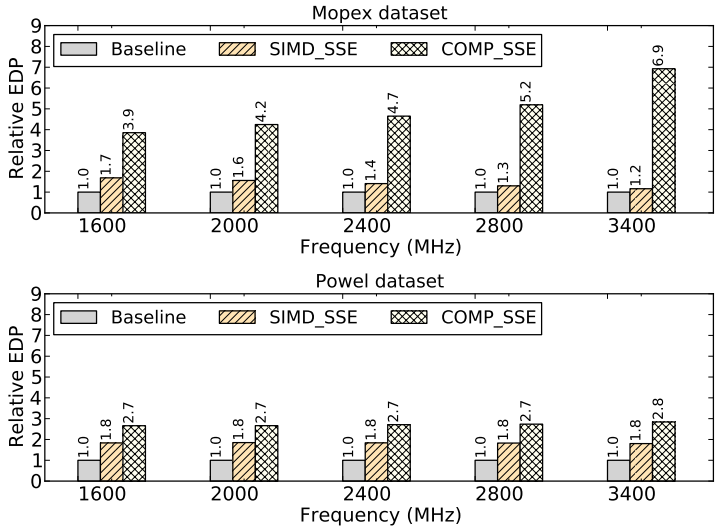


Figure 7: Relative EDP on SB.

In terms of EDP, depicted in the figures Fig. 4c, Fig. 4f and Fig. 10, *Baseline* and *uncompressed SIMD* kernels exhibit lower EDP at higher core frequencies. EDP emphasizes on performance given similar energy consumption. Since the frequency impact on performance for these kernels is limited by the memory subsystem, the effects on EDP are negative. On the other hand, the additional computational requirements of the *COMP* kernels translate into EDP improvements as we increase core frequency. Looking at the relative energy efficiency of different kernels (Fig. 7 and Fig. 9), we find that

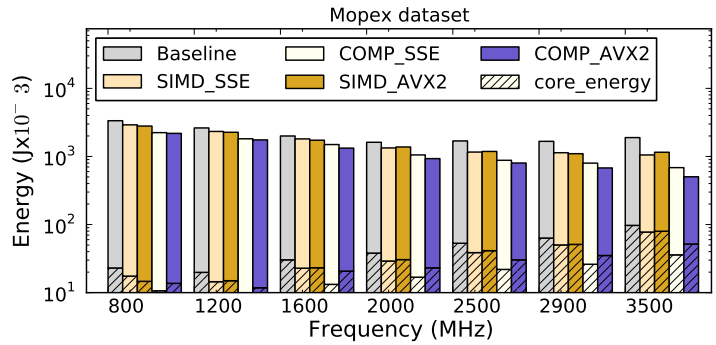


Figure 8: System energy consumption.

Table 3: Average Cache and Memory Related Events on Haswell Processor

Powel kernel	L1 data cache miss rate (%)		L2 cache miss rate (%)		L3 cache miss rate (%)		Total L1 data cache misses		Total L1 data cache accesses		Total L3 cache misses		L1 data prefetch		Total instruction count	
	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M
Baseline	6.40	6.42	22.16	50.48	4.04	20.98	59987609	88373145	3838612	5676306	71741	71741	1642122	9350788	77886930	114747225
SIMD_SSE	20.65	20.74	36.71	62.86	4.44	24.39	18326079	26999387	3784318	5598951	156674	61740	861536	5766500	44983182	66271226
SIMD_AVX2	41.13	41.45	44.11	65.72	8.55	29.13	9163079	13507921	3769232	5599101	156674	144321	1072368	5766500	19993182	29454027
COMP_SSE	19.65	20.34	19.64	29.65	5.97	7.43	8794528	12387811	1728335	2520223	44953	20195	54229	4170770	59734332	83074329
COMP_AVX2	20.07	21.90	21.63	36.18	6.78	9.60	6002619	8119767	1727529	2524005	44953	25315	87837	4170770	38332660	52757265

P=*Powel dataset* M=*MOPEX dataset*

the *COMP* kernels are more energy efficient in all respects. It is also clear that AVX2 instructions do not provide any significant energy efficiency gain over the SSE instructions for *uncompressed SIMD* kernels. The memory subsystem becomes a bottleneck and cannot provide data fast enough to feed the AVX2 256-bit registers. In contrast, the improvements on total data-elements loaded per cache access of the *COMP* kernels is enough to make AVX2 beat SSE in terms of performance and energy.

It is also interesting to note that the system energy consumption is significantly dominated by the uncore part of the system, especially at low core frequencies (Fig. 8). The memory-boundedness nature of the application makes core power dissipation small, far from its design limits (TDP). Moreover, the core-energy consumption increases for the *Baseline* and uncompressed SIMD kernels with core frequency, but the additional frequency does not help to reduce the system energy consumption. As the *COMP* kernels use the idle CPU cycles to reduce the memory traffic, the system energy consumption is reduced at high core frequencies.

6.4 Memory statistics

We have established that the performance of the *Baseline* kernel in Fig. 4 is limited by the memory hierarchy at a high core frequencies. To validate this assumption and identify the root of this behavior, we analyze different micro-architectural properties of the system, such as the percentage of stalled CPU cycles due to lack of resources, including Re-Order Buffer (ROB) hazards, unavailable Reservation Station (RS) slots, unavailable Load/Store Queue (LSQ) slots, and contention for Floating Point (FP) units. The result is presented in Fig. 11. From the figure, we observe that approximately 44% of the CPU cycles are stalled for resource related reasons and the memory stalls account for 34% at the lowest core frequency. But as the core frequency increases, the percentage of stalled-cycles increases, eventually reaching $\approx$59%. The memory resources become the most dominant contributor of the stalled CPU cycles (approximately 57.6% at the highest core frequency), and account for the limited performance when scaling frequency.

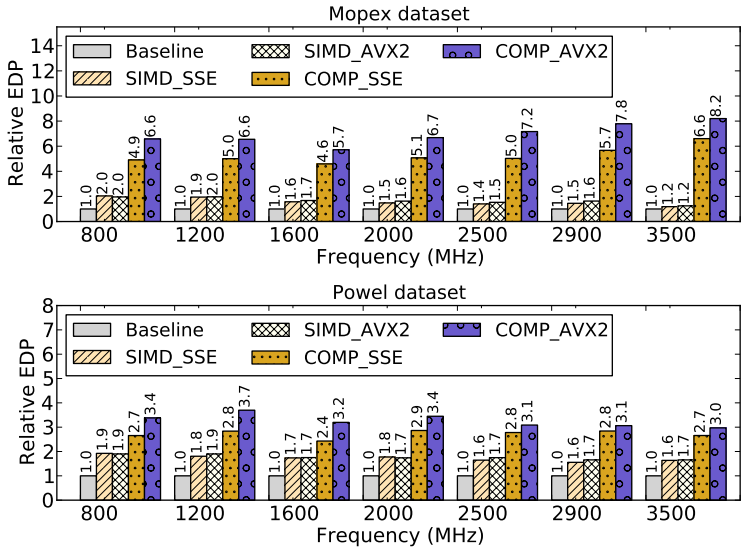


Figure 9: Relative EDP on HL.

In addition, SIMD computations increase the pressure on the memory subsystem, increasing CPU stalls up to 80% of the total stalled CPU cycles. For this reason, SIMD extensions of the Baseline kernel (i.e. *uncompressed SIMD*) are not able to make the best use of the SIMD hardware.

In contrast to the *Baseline* and *uncompressed SIMD* kernels, the *COMP* kernels show much better performance and energy efficiency. We argue that the integration of V-PFORDelta coding helps to to minimize the stall time

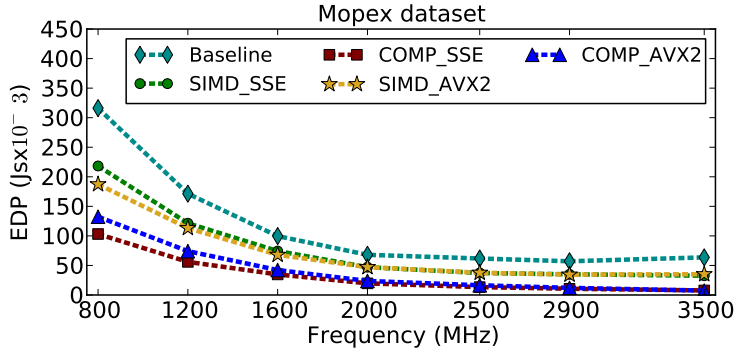


Figure 10: System EDP on HL.

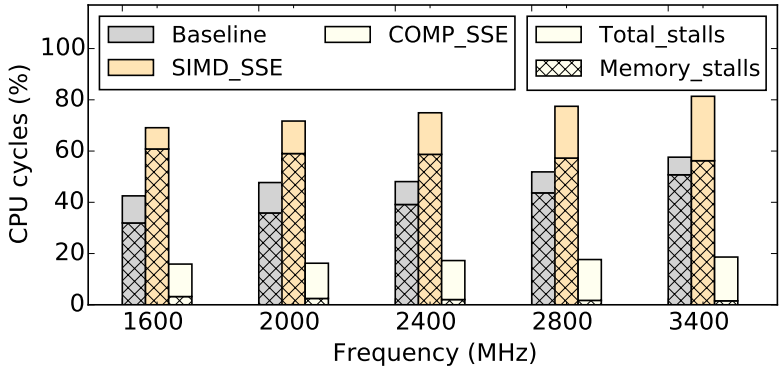


Figure 11: Percentage (%) of CPU cycles stalled due to memory resource limitation on SB system for Powel dataset.

due to memory hierarchy by reducing the amount of data need to be transferred along the memory hierarchy. Additionally, the use of SIMD prefetch instruction to load the data from the main memory to the L1 data cache ahead of their expected usage also helps to improve the data locality property of the *COMP* kernels. To support these claims, we present the statistics of memory related events in Table 3 and Table 4. In Table 3, we can observe that the *COMP_SSE* kernel reduces the total number of L1 data cache accesses (and misses) down to half of the number of accesses (and misses) required by the *SIMD_SSE* kernel on SB system. This eventually reduces the percentage of memory access waiting time down to 2% (from 33% on Powel dataset at peak frequency on SB) of the total CPU cycles and leads to significant performance and energy efficiency improvements. Additionally, SIMD prefetching also helps to improve performance by around 2%.

In Table 4, we present the results of different types of cache misses obtained from DineroIV cache simulator. From the figure, we can see that, most cache misses are due to compulsory and capacity misses for the *Baseline* kernel. This is because the time series dataset is fairly large, and it exhibits poor

Table 4: Cache-miss Statistics from DineroIV Cache Simulator for part of the MOPEX Dataset

Kernel	L1 cache misses				L2 cache misses				L3 cache misses			
	total	compulsory	capacity	conflict	total	compulsory	capacity	conflict	total	compulsory	capacity	conflict
Baseline	1827136	652506	1111351	63279	1759536	652506	1106856	174	1758535	652506	1105978	51
SIMD_SSE	1947530	652509	1233761	61260	1880533	652509	1227791	233	1822096	652509	1169406	181
COM_SSE	1281381	342123	934314	4944	1270162	342123	927883	156	516574	342123	153377	54

temporal locality. Since the *COMP_SSE* kernel uses V-PFORDelta coding technique to operate with the compressed dataset, the number of compulsory and capacity cache misses is reduced significantly. It is also important to note that conflict misses are also reduced for the *COMP_SSE* kernel. Since the compressed data structure in the *COMP_SSE* kernel is designed to place the data involved in a certain computation in the same cache block, it reduces unnecessary cache eviction and increases the cache block utilization.

6.5 Compression ratio and decompression speed

In the last but equally important study, we emphasize on comparing the compression ratio and decompression rate of V-PFORDelta coding with several other integer compression methods. To conduct this experiment, we use (v_i) vectors from both Powel and Mopex datasets as input data. The results are presented in Table 5.

As expected, Simple9 provides better compression ratio as well as decompression rate compared to the VarByte coding scheme for both datasets. However, Frame-of-reference (e.g SimpleFOR, FastPFOR, SIMDFastPFOR) coding schemes appear to be more suitable for the time series data compression as the differences among the consecutive data points in each time series dataset are small. Nevertheless, V-PFORDelta coding outperforms the aforementioned coding methods in terms of decoding speed by at least (3.5% – 6.0%) depending on the dataset used as input. As we opted to trade-off compression ratio in favor of decompression speed, unlike the block-based compression algorithms (e.g. SIMDFastPFOR), V-PFORDelta keeps the bit-length of the delta-bits same over the complete dataset so as to facilitate on-the-fly decompression.

7 Conclusion

This paper presents V-PFORDelta – a SIMD-based differential compression method for cache/memory bound applications. V-PFOR-Delta not only reduces the amount of data to be transferred along the memory hierarchy, but also reduces data storage requirements and minimizes total cache accesses and misses, leading to overall performance and energy efficiency improvements. Furthermore, the effectiveness of V-PFORDelta is demonstrated using SSE and AVX2-based SIMD computations on an industrial hydrological time series application. The experimental results show that the SSE-based implementation can improve the performance and energy efficiency of the application up to 2.8 and 6.6 times, respectively. AVX2-based implementation achieves even better results by improving the performance and energy

Table 5: Results of Compression Ratio and Decompression Speed on Sandy Bridge Processor Using Powel and Mopex Time Series Dataset.

Compression Method	**Powel Dataset**		**Mopex Dataset**	
	bits/int	Mint/sec	bits/int	Mint/sec
VarByte	13.23	626	8.98	706
Simple9	10.17	898	5.56	882
VarIntGB	14.25	914	10.74	1260
SimpleFOR	11.12	1285	4.89	984
FastPFOR	11.12	1340	5.10	1140
SIMDFOR	32.00	1425	32.00	1430
SIMDSimpleFOR	11.12	1562	4.89	1179
SIMDFastPFOR	11.12	1718	5.10	1503
V-PFORDelta	15.18	1776	9.32	1598

Mint=*Millions of integers*

efficiency up to 3.1 and 8.2 times, respectively, on different Intel multi-core systems. The compressed kernels benefit more from frequency scaling than the baseline code, that no longer benefits after one GHz (in energy terms).

In future, we would like to make a fine-grain multi-threaded implementation of the proposed method and evaluate its effectiveness on many-core platform (e.g. Xeon Phi).

References

[1] Sparsh Mittal and Jeffrey Vetter. 'A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems'. In: *IEEE Transactions on Parallel and Distributed Systems* 99.1 (2015), pp. 1–14.

[2] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang and Yan Solihin. 'Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling'. In: *Proceedings of the International Symposium on Computer Architecture*. 2009, pp. 371–382. ISBN: 978-1-60558-526-0.

[3] Ma Lei, Luan Shiyan, Jiang Chuanwen, Liu Hongling and Zhang Yan. 'A Review on the Forecasting of Wind Speed and Generated Power'. In: *Renewable and Sustainable Energy Reviews* 13.4 (2009), pp. 915–920.

[4] Tim Bollerslev. 'A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return'. In: *The Review of Economics and Statistics* 69.3 (1989), pp. 542–547.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann Publishers Inc., 2012.

[6] Juan M. Cebrián, Magnus Jahre and Lasse Natvig. 'Optimized Hardware for Suboptimal Software: The Case for SIMD-aware Benchmarks'. In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. Mar. 2014, pp. 66–75.

[7] Matthias Boettcher, Bashir M. Al-Hashimi, Mbou Eyole, Giacomo Gabrielli and Alastair Reid. 'Advanced SIMD: Extending the Reach of Contemporary SIMD Architectures'. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. 2014, pp. 1–4.

[8] Davendar Kumar Ojha and Geeta Sikka. 'A Study on Vectorization Methods for Multicore SIMD Architecture Provided by Compilers'. In: *Advances in Intelligent Systems and Computing* 248.1 (2014), pp. 723–728. ISSN: 978-3-319-03107-1.

[9] Changkyu Kim, Nadathur Satish, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar and Pradeep Dubey. *Technical Report: Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology*. 2012.

[10] Amanda Svensson Marvel. *Memory Layout Transformations*. 2013. URL: https://software.intel.com/en-us/articles/memory-layout-transformations.

[11] Keith E Mathias and L. Darrell Whitley. 'Changing Representations During Search: A Comparative Study of Delta Coding'. In: *Evolutionary Computation* 2.3 (1994), pp. 249–278.

[12] Hugh E. Williams and Justin Zobel. 'Compressing Integers for Fast File Access'. In: *The Computer Journal* 42.3 (1999), pp. 192–201. DOI: 10.1093/comjnl/42.3.193.

[13] PG Howard and JS Vitter. 'Arithmetic Coding for Data Compression'. In: *Proceedings of the IEEE* 82.6 (1994), pp. 857–865. ISSN: 0018-9219. DOI: 10.1109/5.286189.

[14] Jiangong Zhang, Xiaohui Long and Torsten Suel. 'Performance of Compressed Inverted List Caching in Search Engines'. In: *Proceedings of the International Conference on World Wide Web*. Apr. 2008, pp. 387–396. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367550.

[15] Jonathan Goldstein, Raghu Ramakrishnan and Uri Shaft. 'Compressing Relations and Indexes'. In: *Proceedings of the International Conference on Data Engineering*. Feb. 1998, pp. 370–379.

[16] Benjamin Schlegel, Rainer Gemulla and Wolfgang Lehner. 'Fast Integer Compression using SIMD Instructions'. In: *International Workshop on Data Management on New Hardware*. June 2010, pp. 34–40.

[17] Alexander A. Stepanov and Anil R. Gangolli. 'SIMD Based Decoding of Posting Lists'. In: *Proceedings of the International Conference on Information and Knowledge Management*. Oct. 2011, pp. 317–326.

[18] Naiyoung Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu and Sheng Lin. 'Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units'. In: *Proceedings of the VLDB Endowment*. Vol. 4. Sept. 2011, pp. 470–481.

[19] Xudong Zhang, Wayne Xin Zhao, Dongdong Shan and Hongfei Yan. 'Group-Scheme: SIMD-based Compression Algorithms for Web Text Data'. In: *Proceedings of the International Conference on BigData*. Oct. 2013, pp. 525–530. ISBN: 978-1-4799-1292-6.

[20] D. Lemire and L. Boytsov. 'Decoding Billions of Integers Per Second through Vectorization'. In: *Software: Practice and Experience* 45 (Jan. 2015), pp. 1–29.

[21] Daniel Lemire, Leonid Boytsov and Nathan Kurz. 'SIMD Compression and the Intersection of Sorted Integers'. In: *Software: Practice and Experience* (Apr. 2015).

[22] *The Power of Powel*. URL: http://www.powel.com/About-Powel/.

[23] Q. Duan, J. Schaake, V. Andreassian, S. Franks, G. Goteti, H.V. Gupta, Y.M. Gusev, F. Habets, A. Hall, L. Hay, T. Hogue, M. Huang, G. Leavesley and X. Liang. 'Model Parameter Estimation Experiment (MOPEX): An Overview of Science Strategy and Major Results from the Second and Third Workshops'. In: *Journal of Hydrology* 320 (2006), pp. 3–17.

[24] Intel. *Intel 64 and IA-32 Architectures Software Development Manual*. Dec. 2011. URL: http://download.intel.com/products/processor/manual/325462.pdf.

[25] R. James. *Additional AVX-512 Instructions*. 2014. URL: https://software.intel.com/en-us/blogs/additional-avx-512-instructions.

[26] Hallgeir Lien, Lasse Natvig, Abdullah Al Hasib and Jan Christian Meyer. 'Case Studies of Multi-core Energy Efficiency in Task Based Programs'. In: *Proceedings of the International Conference on ICT as Key Technology against Global Warming*. Vol. 7453. Sept. 2012, pp. 44–54.

[27] *Performance Application Programming Interface*. URL: http://icl.cs.utk.edu/papi/index.html.

[28] Jan Edler. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. URL: http://www.cs.wisc.edu/~markhill/DineroIV.

[29] Valgrind. *Lackey: An Example Tool*. URL: http://valgrind.org/docs/manual/lk-manual.html.

Paper C.1

Energy Efficiency Effects of Vectorization in Data Reuse Transformations for Many-core Processors – A Case Study

Abdullah Al Hasib, Lasse Natvig, Per Gunnar Kjeldsberg and Juan M. Cebrián

Journal of Low Power Electronics and Applications, 2017

Abstract

Thread-level and data-level parallel architectures have become the design of choice in many of today's energy-efficient computing systems. However, these architectures put substantially higher requirements on the memory subsystem than scalar architectures, making memory latency and bandwidth critical in their overall efficiency. Data reuse exploration aims at reducing the pressure on the memory subsystem by exploiting the temporal locality in data accesses. In this paper, we investigate the effects on performance and energy from a data reuse methodology combined with parallelization and vectorization in multi- and many-core processors. As a test case, a full-search motion estimation kernel is evaluated on Intel® Core™ i7-4700K (Haswell) and i7-2600K (Sandy Bridge) multi-core processors, as well as on an Intel® Xeon Phi™ many-core processor (Knights Landing) with Streaming Single Instruction Multiple Data (SIMD) Extensions (SSE) and Advanced Vector Extensions (AVX) instruction sets. Results using a single-threaded execution on the Haswell and Sandy Bridge systems show that performance and EDP (Energy Delay Product) can be improved through data reuse transformations on the scalar code by a factor of $\approx 3x$ and $\approx 6x$, respectively. Compared to scalar code without data reuse optimization, the SSE/AVX2 version achieves $\approx 10x/17x$ better performance and $\approx 92x/307x$ better EDP, respectively. These results can be improved by 10% to 15% using data reuse techniques. Finally, the most optimized version using data reuse and AVX512 achieves a speedup of $\approx 35x$ and an EDP improvement of $\approx 1192x$ on the Xeon Phi system. While single-threaded execution serves as a common reference point for all architectures to analyze the effects of data reuse on both scalar and vector codes, scalability with thread count is also discussed in the paper.

1 Introduction

The continuously-increasing computational demands of advanced scientific problems, combined with limited energy budgets, has motivated the need to reach exascale computing center systems under reasonable power budgets (below 20 MW) by the year 2020. Such systems will require huge improvements in energy efficiency at all system levels. Indeed, system efficiency needs to increase from the current 33 Pflops/17 MW (515 pJ/op) to 1 Eflops/20 MW (20 pJ/op) [1]. Most performance and energy improvements will come from heterogeneity combined with coarse-grained parallelism, through Simultaneous Multithreading (SMT) and Chip Multiprocessing (CMP), as well as fine-grained parallelism, through Single Instruction Multiple Data (SIMD) or vector units.

Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are SIMD instruction sets supported by Intel. SSE, AVX and AVX512 support 128-bit, 256-bit and 512-bit vector computations respectively. Additionally, NEON and SVE in ARM, AltiVec in Motorola and IBM and 3DNow! in AMD are examples of SIMD instruction sets to enable vectorization in different platforms. Many applications can potentially benefit from using SIMD instructions for better performance, higher energy efficiency and greater resource utilization [2]. However, modern compilers do not yet have adequate auto-vectorization support for complex codes to maximize the potential of SIMD instructions [3, 4]. Consequently, when code efficiency is required, the vectorization is often written manually in assembly language or using SIMD intrinsics.

In addition, while processor performance has increased significantly, the memory subsystem has not improved at the same rate. SIMD architectures push the limits of the memory subsystem even further, since now, a single instruction works with N data elements simultaneously, which need to be transferred along the memory hierarchy. Such architectures can even turn CPU-bound applications into memory-bound, depending on the vector length and the arithmetic intensity of the computations (seen as floating point operations per byte of data). Improvements in memory latency/bandwidth (the amount of data that can be accessed in a given period of time), reduction of the energy cost when accessing the data, alongside with better data reuse strategies, are now considered as the key challenges in contemporary computer architectures.

For data-dominated and memory-bound applications such as multimedia algorithms, the Data Transfer and Storage Exploration (DTSE) is a complete

methodology for obtaining and evaluating a set of data reuse code transformations [5]. The fundamental idea behind these transformations is to systematically move data accesses from background memories to smaller foreground memory blocks using less energy. Smaller memories can have reduced access latency, while dissipating less power. This approach eventually results in significant improvements in performance and energy savings.

To sum up, it is becoming crucial to exploit both parallelization via multithreading and through the use of vectorization (i.e., multilevel parallelism) to achieve the required performance on current and future multi-core and many-core processors that run memory bound applications. This paper presents a case-study of the effects of multilevel parallelism combined with the DTSE methodology. The contributions of the paper can be summarized as follows:

- We present a set of data reuse transformation strategies and evaluate the effect of data-parallelism using vectorization on these transformation methodologies on a single core. We further extend our study by analyzing the effects of parallelism at different granularities by combining vectorization with multithreading. For coarse-grained parallelism, the OpenMP parallel programming model is used to provide multithreading across multiple cores. On the other hand, SSE-/AVX-based vectorization on each core is used for fine-grained data parallelism.
- We consider both multi-core and many-core system architectures in our study. For multi-core architecture, Intel Sandy Bridge and Haswell multi-core CPUs are used. For many-core architecture, the Intel Xeon Phi Knights Landing (KNL) processor is used. A full-search motion estimation kernel is used as a test application.

This paper is organized as follows: Section 2 describes related work. Section 3 presents the applied methodology to improve energy efficiency, and 4 illustrates the use of the methodology on the motion estimation algorithm. Section 5 presents and discusses our results. Finally, we conclude the paper in Section 6.

2 Related Work

A large body of research on data reuse exploration methodologies for multimedia applications has led to numerous approaches for improving memory latency and reducing their energy footprint. Wuytack et al. [5] presented

almost 20 years ago a formalized methodology for data reuse exploration. The idea is to exploit temporal locality of memory accesses using an optimized custom memory hierarchy. It is taken further and oriented towards predefined memory organization in [6]. The authors in [7, 8] study the effects of data reuse decisions on power, performance and area in embedded processing systems. The effect of data reuse transformations on a general purpose processor has been explored in [9]. In [10], the authors presented the effect of data reuse transformations on multimedia applications using an application-specific processor. The research described so far focuses on single-core systems and mainly relies on simulation-based modeling when estimating energy efficiency.

In [11], Kalva et al. studied parallel programming on multimedia applications, but lacked energy-efficiency analysis. In [12], Chen et al. presented different optimization opportunities of the Fast Fourier Transform (FFT) to improve performance on the IBM Cyclops-64 chip architecture. In [13], Zhang et al. studied both intra-core and inter-core data reuse and presented a technique to exploit all of the available cores to boost overall application performance. In [14], He et al. proposed a bilinear quarter approximation strategy for fractional motion estimation design together with a data reuse strategy for ultrahigh definition video applications. Lifflander et al. in [15] presented a work-stealing algorithm for fork-/join-based parallel programming models to gain performance boost-up though improving the data locality property.

These earlier studies on parallel architectures emphasized performance rather than energy efficiency. In contrast, our work is not limited to performance analysis only; rather, it is extended by energy efficiency analysis on three different state-of-the-art multi- or many-core processors, namely Sandy Bridge and Haswell CPUs and the Xeon Phi KNL processor.

In [16], Marchal et al. presented an approach for integrated task-scheduling and data-assignment for reducing SDRAM costs for multithreaded applications. However, this work did not couple the data reuse analysis with a parallel programming model the way we do here. Here, we extend our previous work presented in [17], where we analyzed the effect on the performance and energy efficiency of coupling multithreaded parallelism with the data reuse transformations. However, [17] was limited to a single multi-core platform, and the effect of fine-grained data-parallelism through vectorization was not covered.

Data reuse techniques (mainly blocking) and other low level optimizations

for Intel's many-core architectures are covered in [18]. A combination of programming language features, compiler techniques and operating system interfaces are presented in [19] that can effectively hide memory latency for the processing lanes. Their study is based on the active memory cube system, a novel heterogeneous computing system, substantially different from our test platforms. Dong et al. in [20] made a custom implementation of linear algebra kernels for GPUs to obtain ≈10% power savings for a hydrometric kernel. In [21], the authors presented a micro-architectural technique to approximate load values on load misses so as to reduce the cost of memory accesses. In [22], the authors provided emphasis on making efficient use of hardware vector units in order to achieve optimal performance on multi- and many-core architectures. They also demonstrated the limitations of auto-vectorization over hand-tuned intrinsic-based vectorization for the applications with irregular memory accesses on a Xeon Phi co-processor. Furthermore, the authors in [23] argued for Cray-style temporal vector processing architectures as an attractive means of exploiting parallelism for the future high performance embedded devices.

Though these articles demonstrated the importance of vector processing in achieving better performance as we do in this paper, they do not provide energy efficiency or DTSE-like analysis for a full-search motion estimation or similar kernels.

3 Energy Efficient Methodology for Multi-core and Many-core Processor

The approach presented in this paper is a combination of three techniques: the data reuse transformation methodology, parallelization by dividing computational work on several cores and vectorization by using the SIMD instructions available in each core. Note that vectorization is also a technique for parallelization. For example, an application divided onto four cores each using a four-way vectorization exhibits, in total, a 16-way parallelization.

3.1 Approaches to Improve Performance and Energy Efficiency

Data Reuse Transformation

The most central concept of a data reuse transformation is the use of a memory hierarchy that exploits the temporal locality of the data accesses [5]. This memory hierarchy consists of smaller memories where copies of data from larger memories that expose high data reuse are stored. For data-intensive applications, this approach causes significant energy savings since

smaller memories built on similar technology consume less energy per access and have significantly shorter access times. In addition, the average access latency to data is reduced, since the transformations act as a complex software prefetching mechanism.

Parallelization

Multi-core processors can potentially achieve higher performance with lower energy consumption compared to uni-processor system. However, there are challenges in exploiting the available hardware parallelism without adding too much overhead. Different parallel programming models have been developed for speeding up applications by using multiple threads or processes [24]. Parallel programs can be developed by using specific programming models (e.g., OpenMP, OmpSs, Cilk) [25] or can use explicit threading support from the operating system (e.g., POSIX (Portable Operating System Interface) threads).

Vectorization

Vectorization using SIMD constructs is very efficient in exploring data level parallelism since it executes multiple data operations concurrently using a single instruction. Therefore, SIMD-computations often result in significant performance improvements and reduced memory accesses, eventually leading to higher energy efficiency for an application [26]. However, extracting performance from SIMD-based computations is not trivial as it can be significantly affected by irregular data access. In multi-core programming memory bandwidth often becomes a critical bottleneck, as the data movement from memory to the register bank increases with both vectorization and parallelization [2]. On the other hand, memory latency can sometimes be hidden by optimization techniques, such as SIMD prefetching or improved data locality (e.g., cache blocking).

3.2 Data Reuse Transformations with Parallelization and Vectorization

In our case-study, we combine the aforementioned approaches to further optimize the performance and energy efficiency of data reuse transformations. Our approach consists of the following main steps.

- In the first step, the sequential optimized algorithm is translated into a parallel algorithm that follows a typical parallel workflow:
  - The original workload is divided into a number of p smaller sub-workloads. Each of the p sub-workloads is assigned to a thread

for completion.

- Multiple threads are executed simultaneously on different cores, and each thread operates independently on its own sub-workload.

- When all of the sub-workloads are completed by the threads, an implicit barrier ensures that all of the local results of the completed sub-workloads are combined together through a reduction step.

At this stage, coarse-grained parallelism is applied through multithreading/simultaneous multithreading on multiple cores.

- Next, to exploit the data parallelism support available in the systems, we use SIMD computations on each thread. Since modern compilers still do not have adequate auto-vectorization support for complex codes [3, 4], we use the SSE, AVX2 and AVX512 intrinsics to manually perform the SIMD computations. We ensure that the data layout of the demonstrator application is properly aligned with the boundaries of the vector registers. SIMD prefetching is used to copy the copy-candidates described below into smaller buffers to hide the latency of the memory transfer.

- In the final step, we apply the proposed model for data reuse transformations as presented in [5]. Here, we identify the datasets that are reused multiple times within a short period of time. These are called copy-candidates. In this study, the copy-candidate tree from [5] is used for the scalar version of the demonstrator kernel, whereas it has been slightly changed when SIMD computations are used. Depending on their size and position in the tree, the copy-candidates are mapped into appropriate layers of the system's memory hierarchy. To map a data block into its layer, we use the SIMD-prefetch intrinsic (i.e., *_mm_prefetch*) with the appropriate hint options for both scalar and vectorized codes. Generally, *_MM_HINT_T0*, *_MM_HINT_T1* and *_MM_HINT_T2* options can be used for prefetching data into L1, L2 and L3 caches, respectively [27]. Compiler-assisted software prefetching is disabled to avoid auto-prefetching of data from memory. Furthermore, we also use SIMD stream intrinsics (*_mm_stream_load* and *_mm_stream*) in our vectorized codes to limit cache pollution due to movement of data across the memory hierarchy.

4 Demonstrator Application: Full-Search Motion Estimation Algorithm

To evaluate the efficiency of the combined approach, we have used a full-search motion estimation algorithm as a case-study. In this section, we use it to describe the main steps of the approach.

Algorithm 1 Motion estimation kernel (source: [28, 5]).
Input: Current frame (*Cur*), Reference frame (*Ref*), frame Height (*H*), frame Width (*W*), search range (*m*), block size (*n*).
Output: motion vector (Δ_{opt}).

```
1:  for g=0; g<H/n; g++ do                          ▷ Vertical current block counter
2:      for h=0; h<W/n; h++ do                      ▷ Horizontal current block counter
3:          Δopt[g][h] = Δopt[g][h] + ∞
4:          for i= -m; i<m; i++ do                  ▷ Vertical searching of reference window
5:              for j= -m; j<m; j++ do              ▷ Horizontal searching of reference window
6:                  Δ = 0
7:                  for k=0; k<n; k++ do            ▷ Vertical traversal of MB
8:                      for l=0; l<n; l++ do        ▷ Horizontal traversal of MB
9:                          Δ = Δ + | Cur[g×n+k][h×n+l] - Ref[g×n+i+k][h×n+j+l]) |
10:                     end for
11:                 end for
12:                 Δopt[g][h] = min(Δ, Δopt[g][h])
13:             end for
14:         end for
15:     end for
16: end for
```

4.1 Sequential Motion Estimation Algorithm

Motion Estimation (ME) is used in most video compression algorithms as it can be used to find temporal redundancy in a video sequence. In a block-matching ME algorithm, a video frame ($W \times H$) is divided into a number

of ($n \times n$) non-overlapping blocks called Macro-Blocks (MBs), where the value of n can be {2,4,8,16, ...}. Next, for each MB in the Current frame (*Cur*), a matching MB within the search area ($m \times m$) in the Reference frame (Ref) is selected having the least cost value. There are several possible cost functions. The Sum of Absolute Difference (SAD) is widely used and computationally inexpensive and so selected for this study. The SAD between an MB in *Cur* and an MB in Ref is given by,

$$SAD = \sum_{i=1}^{i=n} \sum_{j=1}^{j=n} | Cur_{ij} - Ref_{ij} | \quad (1)$$

where Cur_{ij} and Ref_{ij} are the pixel values at position (i, j) in the MBs of Cur and Ref, respectively.

Finally, the relative displacement of the current MB and its matching MB is the motion vector. A full-search ME algorithm performs an exhaustive search over the entire search region to find the best match. Running this algorithm is computationally intensive and can account for as much as 80% of the encoding time [11]. This computationally-intensive algorithm exhibits the properties of data reuse and parallelization and is therefore well suited as a test application for our approach.

The full-search ME kernel used in our work is presented in Algorithm 1 [28, 5]. The implementation consists of a number of nested loops. The basic operation at the innermost loop consists of an accumulation of pixel differences (i.e., the SAD computation), while the basic operation two levels higher in the loop hierarchy consists of the calculation of the new minimum. In our experiments, we use the parameters of the QCIF (Quarter Common Intermediate Format) format ($W = 176$, H = 144, m = 16, $n = 8$) on the multi-core system and the Full HD format ($W = 1920$, H = 1080, $m = 16$, $n = 8$) on the many-core platform [29]. In the remainder of this paper, we simply refer to this as the ME application.

4.2 Parallel ME Application across Multiple Cores

In the process of ME algorithm parallelization, we design a multi-core computing model to exploit the data parallelism properties inherent in the ME algorithm. In the algorithm, there is no data dependency between two macro-blocks' SAD. Thus, the SADs of different macro-blocks can be computed in parallel. Considering this, we divide *Cur* into the same number of subsections as there are threads and assign each subsection to a particular thread. Eventually, all of the threads run in parallel across multiple cores to

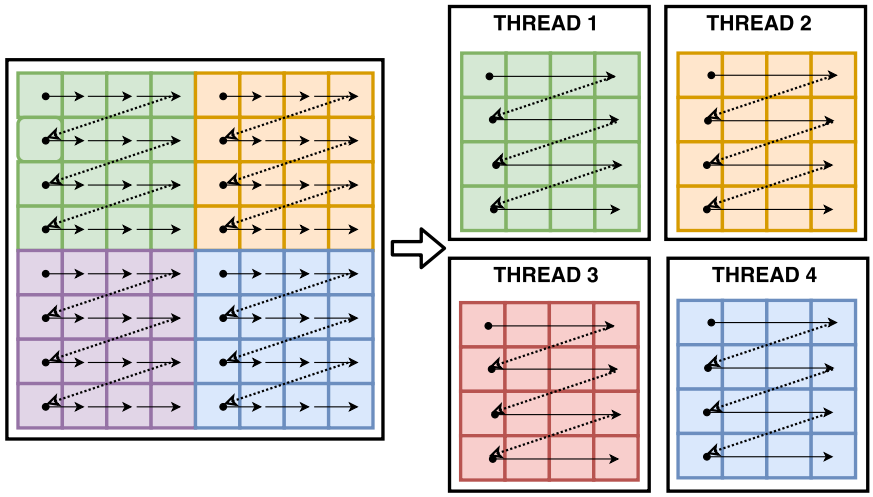


Figure 1: Illustrative motion estimation multi-core computation model. First, a frame is divided into four subsections, each of which consists of four macro-blocks of a size of 4×4 pixels. Then, a thread is assigned for each subsection, and the Sum of Absolute Difference (SAD) computations are done in parallel by multiple threads on multiple cores.

execute the SAD processing. The benefit of this division into subsections is to make full use of the available cores on the underlying platforms. Figure 1 illustrates a four-threaded computation model for an 8×8 frame.

Algorithm 2 represents our parallel ME algorithm, which shows OpenMP parallel programming constructs used to achieve thread-level parallelism across multiple cores [25]. As we can see, the *#pragma omp parallel for collapse*(*2*) directive of the OpenMP programming model is used at the outermost for-loop. The inclusion of this directive will instruct the compiler to fuse the next two loops into a single loop, increasing the amount of work that can be assigned to all processors (cores) of the system. We have also set the *KMP_AFFINITY* environment variable to bind each thread, i.e., each instance of the for-loop, to a specific core. We further use the *OMP_NUM_THREADS* environment variable to control the number of threads to be created at a particular time. Note that though we have used loop-based work-sharing constructs of OpenMP here, other types of work sharing constructs, such as OpenMP tasking constructs (*#pragma omp parallel, #pragma omp task*) can be used, as well.

Algorithm 2 Optimized parallel motion estimation algorithm.
Input: Current frame (*Cur*), Reference frame (*Ref*), frame Height (*H*), frame Width (*W*), search range (*m*), block size (*n*).
Output: motion vector (Δ_{opt})

1: ***#pragma omp parallel for schedule(static, <chunksize>) collapse(2)***
2: **for** *g=0; g<H/n; g++* **do** ▷ Vertical current block counter
3: **for** *h=0; h<W/n; h++* **do** ▷ Horizontal current block counter
4: **for** *k=0; k<2m+n-1; k++* **do** ▷ Copy-candidates are copied to a smaller memory block
5: **for** *l=0; l<2m+n-1; l++* **do**
6: *Buffer_ref[k][l] = Ref[g×n-m+k][h×n-m+l]*
7: **end for**
8: **end for**
9: **for** *k=0; k<n; k++* **do** ▷ Copy-candidates are copied to a smaller memory block
10: **for** *l=0; l<n; l++* **do**
11: *Buffer_cur[k][l] = Cur[g×n+a][h×n+b]*
12: **end for**
13: **end for**
14: $\Delta_{opt}[g][h] = \Delta_{opt}[g][h] + \infty$
15: **for** *i=0; i<2m-1; i++* **do** ▷ Vertical searching of reference window
16: **for** *j=0; j<2m-1; j++* **do** ▷ Vertical searching of reference window
17: $\Delta = 0$
18: **for** *k=0; k<n; k++* **do** ▷ Vertical traversal of MB
19: **for** *l=0; l<n; l++* **do** ▷ Horizontal traversal of MB
20: Δ += | *(Buffer_cur[k][l]- Buffer_ref[i+k][j+l])* |
21: **end for**
22: **end for**
23: $\Delta_{opt}[g][h] = \min(\Delta, \Delta_{opt}[g][h])$
24: **end for**
25: **end for**
26: **end for**
27: **end for**

Table 1: Required C/C++ intrinsics to implement the Motion Estimation (ME) algorithm. Streaming Single Instruction Multiple Data (SIMD) Extensions (SSE), Advanced Vector Extensions (AVX) 2 and AVX512 are the SIMD instruction sets that operate on 128-bit, 256-bit and 512-bit vector registers, respectively.

Operation	SSE	AVX2	AVX512
bitwise OR	_mm_or_si128	_mm256_or_si256	*_mm512_or_si512*
bitwise AND	_mm_and_si128	_mm256_and_si256	*_mm512_and_si512*
shift right by a number of bytes	_mm_srli_si128	_mm256_srli_si256	*_mm512_srli_si512*
shift left by a number of bytes	_mm_slli_si128	_mm256_slli_si256	*_mm512_slli_si512*
add four 32-bit integers	_mm_add_epi32	_mm256_add_epi32	*_mm512_add_epi32*
shuffle four 32-bit integers	*_mm_shuffle_epi32*	*_mm256_shuffle_epi32*	*_mm512_shuffle_epi32*
compare four 32-bit integers	_mm_cmpgt_epi32	_mm256_cmpgt_epi32	*_mm512_cmpgt_epi32*
store a 128-bit register	_mm_storeu_si128	_mm256_storeu_si256	*_mm512_storeu_si512*
load to 128-bit register	_mm_loadu_si128	_mm256_loadu_si256	*_mm512_loadu_si512*
unpack and interleave 32-bit integers	_mm_unpacklo_epi32	_mm256_unpacklo_epi32	*_mm512_unpacklo_epi32*
convert 8-bit integers to 32-bit integers	_mm_cvtepi8_epi32	_mm256_cvtepi8_epi32	*_mm512_cvtepi8_epi32*
element-by-element bitwise AND on 32-bit integers	-	-	*_mm512_mask_and_epi32*
compare packed 32-bit integers, and store the results in mask vector k	-	-	*_mm512_cmpgt_epi32_mask*
compare packed 32-bit integers, and store the results in mask vector k	-	-	*_mm512_cmplt_epi32_mask*
reduce 32-bit integers in a by addition using mask k	-	-	_mm512_mask_reduce_add_epi32
store a 128-bit register	_mm_stream_load_si128	_mm256_stream_load_si256	*_mm512_stream_load_si512*
store to 128-bit register	_mm_stream_si128	_mm256_stream_si256	*_mm512_stream_si512*
Reduce 32-bit integers by addition using mask k and returns the sum	-	-	*_mm512_mask_reduce_add_epi32*

4.3 Data Parallelism in Each Core through Vectorization

The use of SIMD intrinsics will help to reduce the number of memory accesses and instructions in comparison to the sequential execution on both of the considered architectures. The inner loop of the SAD calculation of one macro-block (SSE/AVX2) or two macro-blocks (AVX512) can be partially executed in parallel by using SIMD instructions. Within each iteration of the loop over all block rows, one complete cache-line of the reference and one of the candidate block are loaded into two separate vector registers, and their row-wise SAD value is accumulated over all rows. The AVX512 and SSE/AVX2 instruction-set extensions of the Intel architecture feature a single instruction, computing the element-wise absolute difference of two vector registers and internally reducing each vector of eight consecutive absolute difference values to one 32-bit element. Finally, the accumulated 32-bit SAD value, which is generated for each row of the current block, needs to be extracted from the vector register into a scalar register. Table 1 lists a set of SSE, AVX2 and AVX512 intrinsics used in the ME algorithm implementations.

4.4 ME Optimization Using Data Reuse Transformation

In the final stage of the optimization process, we integrate data reuse transformation strategies into our optimized ME kernel to gain further performance and energy efficiency improvements. Our study is based on a research work of Wuytack et al., who used the DTSE methodology for data reuse transformations and presented a number of different possible transformations using a copy-candidate tree for the ME algorithm [5]. However, in this study, we only consider the subset of transformations reported to be the more energy efficient in [5] and which also fit to the memory hierarchy of our test systems. Figure 2 presents different possible transformations that are considered to optimize the ME algorithm in this work.

In Figure 2, each branch in the copy-candidate tree corresponds to a potential memory hierarchy for a given data-reuse transformation. The vertical dashed lines in the figure indicate the levels of the hierarchy. Each rectangle in the hierarchy is a copy-candidate and corresponds to a block of data to be stored in memory. Each block is annotated with its size. The highlighted branches in the figure (i.e., the branching option with the blue rectangular boxes) indicate a two-layer memory hierarchy for a possible data reuse transformation in each of the frames. For the reference frame, the hierarchy is comprised of an $H \times W$ block and a $(2m + n - 1) \times (2m + n - 1)$ block memory. These blocks are mapped into the L3 and L2 caches of our test

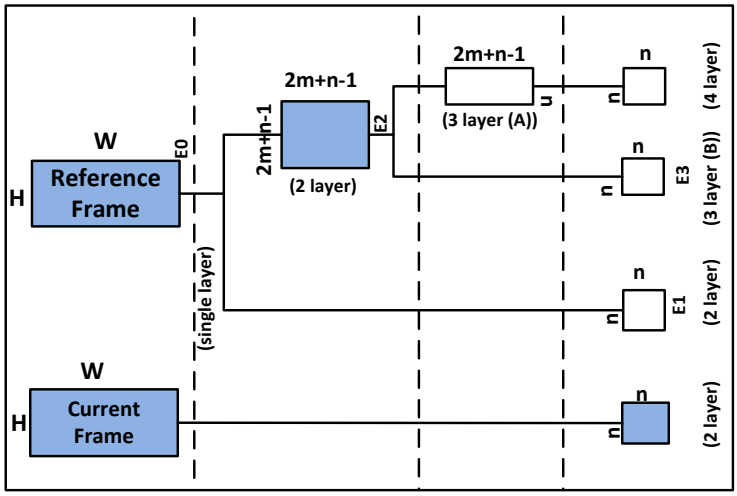


Figure 2: Copy-candidate tree for data reuse decision for the motion estimation algorithm (source: [5]).

platforms. In addition, a two-layer memory hierarchy for the current frame with an $H \times W$ and an $n \times n$ memory block is also introduced. The blocks are mapped into the L3 and L1 caches, respectively. Among the different possible transformation options presented in the copy candidate tree, the options that are labeled with **E** are analyzed in detail in the Results section.

To evaluate the performance and energy efficiency of the different data-reuse transformations presented in Figure 2, the basic ME algorithm (Algorithm 1) has been modified into different versions to exploit different possible transformations. This modification is done by introducing smaller memory blocks (*Buffer_cur, Buffer_ref*) to which the copy-candidates (e.g., each line in the reference window) of the reference frame are copied. The algorithm illustrated in Algorithm 2 presents the most effective solution in terms of performance and energy among the different transformation options we have analyzed in this study. The solution introduces two small buffers for two copy-candidates; one of them is for the Current frame (*Cur*), and the other one is for the Reference frame (*Ref*).

It is also important to note that the copy-candidate tree presented in Figure 2 is used in all of our experiments, except the experiments with AVX512. Architectures supporting AVX512 intrinsics (e.g., KNL) are featured with 512-bit SIMD registers, that operate with 512-bits of data at the same time. Therefore, AVX512 intrinsics access the pixels in two macro-blocks of size

8×8 at the same time. Consequently, a copy candidate of size $n \times n$ becomes less efficient for AVX512-based computations. To address this problem, instead of defining a copy-candidate of size $n \times n$, we define a copy-candidate of size $2n \times n$ for AVX512 computations.

5 Results and Discussion

In this section, we present several experiments to study the effect of vectorization on data reuse transformations. We implement ten different variants of the ME kernel following the data reuse transformation techniques illustrated in Figure 6. The experiment names reflect the different copy-candidates (`E0` through `E3`) in Figure 6. The kernels are implemented in C++ using SSE, AVX2 and AVX512 intrinsics. The ten variants are:

1. *Scalar_E0*: Implementation of the ME kernel (Algorithm 1) without any of the optimization techniques covered in this study (i.e., no data reuse transformations or vectorization). This sequential implementation is the baseline to which we make a comparative study of different kernel implementations.

2. *Scalar_E1*: Non-vectorized ME kernel with data reuse transformation using the $n \times n$ copy-candidate.

3. *Scalar_E2*: Non-vectorized ME kernel with data reuse transformation using the $(2m + n + 1) \times (2m + n + 1)$ copy-candidate.

4. *Scalar_E3*: Non-vectorized ME kernel with data reuse transformation using the $(2m + n + 1) \times (2m + n + 1)$ and $n \times n$ copy-candidates.

5. *SSE_E0*: SSE-vectorized ME kernel without any data reuse transformations.

6. *SSE_E2*: SSE-vectorized ME kernel with data reuse transformation using the $(2m + n + 1) \times (2m + n + 1)$ copy-candidate.

7. *AVX2_E0*: AVX2-vectorized ME kernel without any data reuse transformations.

8. *AVX2_E2*: AVX2-vectorized ME kernel with data reuse transformation using the $(2m + n + 1) \times (2m + n + 1)$ copy-candidate.

9. *AVX512_E0*: AVX512-vectorized ME kernel without any data reuse transformations.

Table 2: Hardware Specifications of the Test Platforms

Processor	**Intel® Core™ i7-2600K**	**Intel® Core™ i7-4700K**	**Intel® Xeon Phi 7250**
Architecture	SandyBridge	Haswell	Knights Landing
Clock Speed	1.6 – 3.4 GHz	0.8 – 3.5 GHz	1.4 Ghz
# of Cores	4 cores / 8 threads		68 cores / 272 threads
L1 Cache	32 KB data + 32 KB inst, 8-way private		
L2 Cache	256 KB, 8-way private		1 MB, 16-way per 2 cores

10. *AVX512_E2*: AVX512-vectorized ME kernel with data reuse transformation using the $(2m + n + 1) \times (2m + n + 1)$ copy-candidate.

5.1 Test System Architecture

This subsection describes the platforms used in our evaluation. It is important to note that since we conduct our experiment on multi- and many-core systems with a memory hierarchy of fixed sized cache-blocks, the copy-candidates are mapped into system caches according to their sizes (Table 2).

Intel® Core™ CPUs

In our experiments, we have used the Intel® Core™ i7-4700K (Haswell) and i7-2600K (Sandy Bridge) processors consisting of four physical cores. Both support Hyper-Threading (HT), which allows the CPU to simultaneously process up to eight threads (i.e., two threads per core). The memory hierarchy consists of a 32-KB Level-1 cache, a 256-KB Level-2 cache and a 8192-KB Level-3 cache. Level-1 and Level-2 caches are private to each core, while the Level-3 cache is shared among the cores. Note that this is a memory hierarchy with a fixed number of levels and sizes, typical for a standard processor. This is different from the assumption in [5], where an application-specific memory hierarchy is assumed. The base clock speeds of the processors are 3.5 GHz and 3.4 GHz, respectively, but it can go as high as 3.9 GHz when Turbo Boost is enabled [27]. However, we disabled dynamic frequency scaling (SpeedStep and Turbo Boost) to get more stable results from the experiments. The systems run under Ubuntu 14.04.3 LTS. All of the kernels are compiled using Intel C++ compiler (ICC Version 14.0.1) with the -O2 option.

Intel Xeon Phi Processor

Xeon Phi: Knights Landing (KNL) is the second revision of Intel's Many Integrated Core Architecture (MIC). Many-core architectures offer a high number of cores (68 in our evaluation platform) with up to four threads per

core and a potential peak performance close to 6 Tflops for single precision floating point. These cores are based on the Silvermont Atom architecture. Cores are out of order and tiled in pairs. Each core contains two Vector Processing Units (VPUs), which work with vector registers up to 512 bits wide. The VPUs are compatible with SSE, AVX/AVX2 and AVX512.

Each tile in the processor shares 1 MB of L2 memory, using a 2D mesh interconnect (or NOC (Network On Chip)) for communication. This interconnect also links the tiles to two DDR4 memory controllers, with a capacity of up to 384 GB and a bandwidth of 90 GB/s. In addition, some KNL models feature a High Bandwidth Memory Multi-Channel DRAM (HBM-MCDRAM), which is accessed using the NOC. This memory is divided into eight stacks, adding up to 16 GB of capacity, and has a bandwidth close to 400 GB/s. The HBM memory can work in different modes, as a scratchpad memory, as an additional cache level or in hybrid mode (combination of the previous two modes).

Our evaluated Xeon Phi platform is based on the Xeon Phi 7250 processor. This processor features 68 cores running at 1.40 GHz. The system runs SUSE Linux Enterprise Server 12 SP1, and the binaries are generated using Intel C++ compiler (Version 17.0.035) with the -O2 optimization level and the -xMIC-AVX512 flag to generate AVX512 code. The devices are configured to work in cache mode, where the HBM acts as an L3.

5.2 Metrics Used for Analysis

We use execution time (in micro-seconds) as the metric for performance evaluation. To estimate on-chip energy consumption, we read the Model-Specific Registers (MSRs) that provide energy measurements for the cores in Haswell and Sandy Bridge [27], since package measurements include the integrated GPU power, and we are not interested in that. For KNL, we measure the whole package energy (including core power and DRAM controller traffic), since the core energy counter is not available in our pre-production system. In addition, we have not found any accurate description of what the DRAM controller actually measures as the measurements might include memory controller and caches, or the accesses to the DDR modules (it should not, since manufacturers can have any brand/technology attached to the system and dissipate different power). We decided to go for the isolated core energy (PP0 i.e. Power-Plane 0) and Package energy (PKG) since both are properly defined. These counters can be accessed either by the RAPL (Running Average Power Limit) interface (root-level) or the powercap interface (user-level). We report both core/package-energy

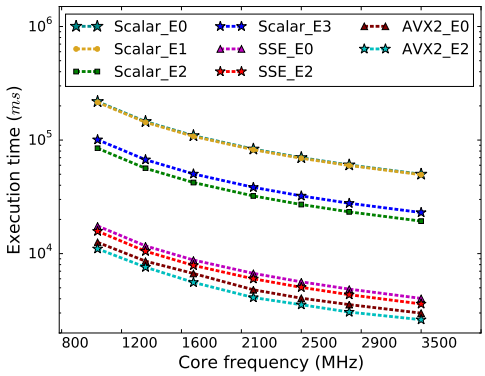
consumption and Energy Delay Product (EDP: *Joule* $\times$ *Second*) [30, 31] to perform energy efficiency analysis. For both, lower values corresponds to better energy efficiency. Speedup, relative on-chip energy and relative EDP at a certain frequency are all computed with respect to the baseline kernel (*Scalar_E0*). The PAPI (Performance Application Programming Interface) [32] is used to track cache- and memory-related events.

5.3 Performance analysis on the Haswell and Sandy Bridge Platforms

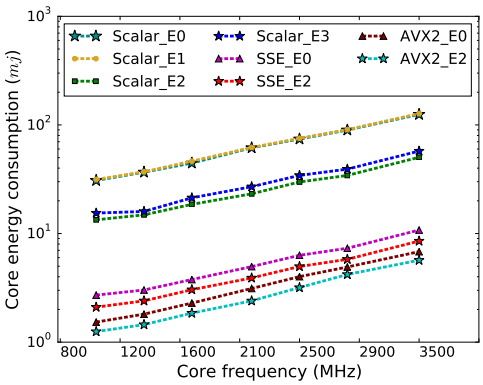
In the first set of experiments, our goal is to gain insight into the effects of vectorization on data reuse transformations. Figure 3 compares the performance of different kernel implementations at different core frequencies on the Haswell platform.

In Figure 3a, we observe a linear impact of core frequency on the performance of the implemented kernels. The kernel execution time decreases with increasing core frequency. The figure also confirms that the performance of the kernel can be improved by using data reuse transformation techniques despite the overhead of copying the copy-candidates into the buffers. Among the three different data reuse transformation techniques (E1, E2 and E3), E2 appeared to be more effective than the other two transformations, and it (i.e., Scalar_E2) provides more than a two-fold performance gain over the unoptimized (i.e., Scalar_E0) solution when integrated with the scalar ME kernel in a single core on the Haswell system. This improvement is attributed to the use of a smaller block size, since a block of $(2m + n - 1) \times (2m + n - 1)$ unsigned-characters corresponds to 2209 ($47 \times 47 \times 1$) bytes, which is less than the Level-1 cache size in our system. Therefore, a full block is brought into the Level-1 cache during computation, which significantly reduces the cost of expensive memory accesses. This cost reduction is evident in the data presented in Table 3, as the total number of stalled CPU cycles on memory subsystems is reduced to one third of the stalled cycles for Scalar_E0. It is also interesting to note that the L1D/L2 cache miss rate for SIMD computations is much higher than the scalar computations, and the cache miss rates increase as the width of SIMD register increases. This observation leaves us room for further optimization, particularly for the systems with wider SIMD registers, such as Scalable Vector Extensions (SVE) [33], which are designed to support up to 2048-bit registers.

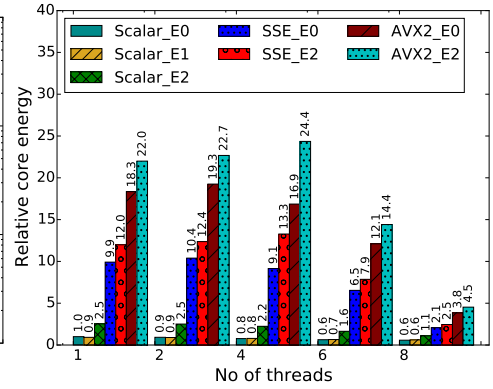
Now, comparing the performance of the vectorized kernels with the scalar kernels, the vectorized kernels clearly outperform the scalar kernels by a significant margin ($\approx$10$\times$ and $\approx$17$\times$ for SSE and AVX2, respectively). This



(a) Execution time



(b) Core energy consumption

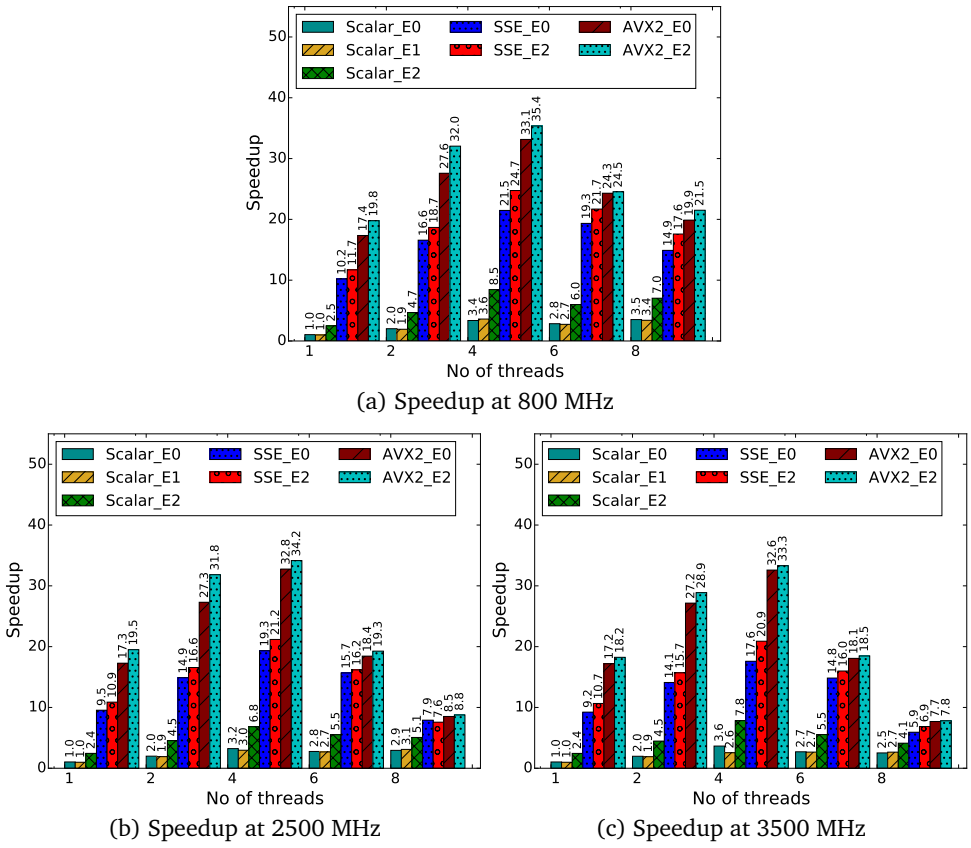


(c) Core energy reduction

Figure 3: Execution time, core-energy consumption and energy efficiency in terms of relative core energy reduction factor (at frequency 3500 MHz) with respect to the core energy consumption of the baseline kernel on Haswell system. (**a**) Execution time; (**b**) core energy consumption; (**c**) core energy reduction.

performance improvement is expected as SIMD computation provides two-fold benefits in this circumstance.

First, it reduces the number of instructions needed to be executed for completing the task by simultaneously processing multiple data points using the vector registers. Since the SSE registers are 128-bit wide, SSE-based SIMD computations can process up to 16 unsigned characters (8-bit) at a time. Therefore, we can potentially achieve 16× performance speedup using SSE-based vectorization in the ME kernel. However, due to the inherent



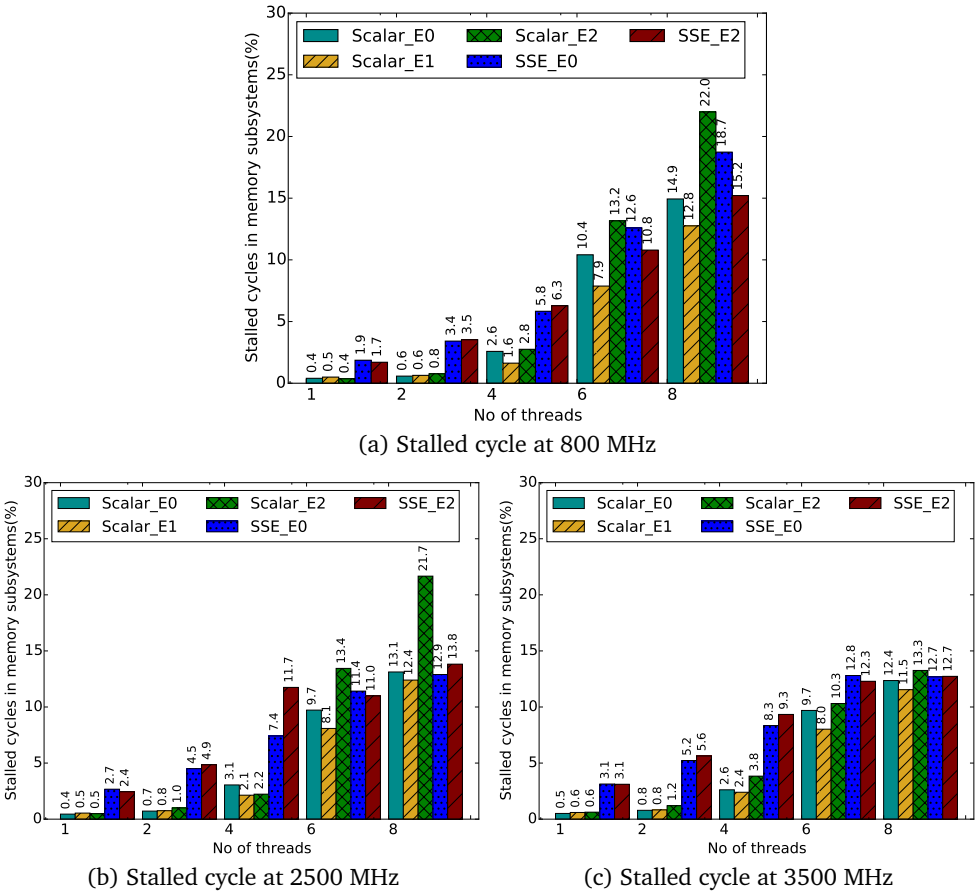
(a) Speedup at 800 MHz

(b) Speedup at 2500 MHz

(c) Speedup at 3500 MHz

Figure 4: Achieved speedup of multithreaded ME-kernel implementations at different core frequencies on Haswell system.

complexity of the algorithm (e.g., 8-bit values need to be converted into 32-bit values to continue further computations), we ended up with ≈10× speedup for *SSE_E0* over *Scalar_E0*. Similarly, using 256-bit SIMD registers for AVX2-based computations, we have achieved speedup ≈17× for *AVX2_E0* over *Scalar_E0*. This speedup is shown in Figure 4, and the reduction of instruction counts are presented in Table 3. Second, the SIMD computation also reduces the required number of cache/memory accesses by simultaneously loading/storing multiple data points using a single load/store operation. This can eventually reduce the CPU waiting time for the memory subsystems and improve the overall system's performance. On the other hand, for bandwidth-bound applications, the increased bandwidth demand caused by the SIMD computations can limit the potential performance im-



(a) Stalled cycle at 800 MHz

(b) Stalled cycle at 2500 MHz

(c) Stalled cycle at 3500 MHz

Figure 5: Percentage of stalled CPU cycles on memory subsystem for multithreaded ME-kernel implementations at different core frequencies on Haswell system.

provements if the memory subsystem cannot sustain such demand. Indeed, the data presented in Table 3 shows that the absolute number of stalled CPU cycles on the memory system for the vectorized ME kernels is reduced, but represents a higher percentage of the overall execution (estimated to be Figure 5).

Another important observation is the fact that the impact of data reuse transformation is greater for the performance of the scalar ME than for the vectorized ME. As shown in Figure 4, the data reuse transformations provide two-fold performance gain as we move from E0 to E2 optimizations

Table 3: Average execution time and energy consumption of sequential kernel implementations at a peak core frequency on the Haswell system. EDP, Energy Delay Product.

ME kernel	Execution time	Stalled CPU cycles	Total CPU cycles	Core energy consumption	EDP	Instruction count	Relative Energy		Cache Miss Rate (L2)
Scalar_E0	48480	564838	164147540	132	6256000	631036879	1	1	28%
Scalar_E1	50745	403386	165314558	136	6292469	631292899	1	1	21%
Scalar_E2	20491	120485	67881181	49	966100	187288526	3	2	12%
Scalar_E3	23008	1828712	80570952	57	1311456	199778723	2	2	11%
SSE_E0	5422	148615	14696743	13	44341	54934069	11	9	35%
SSE_E2	4597	150375	12574005	9	32445	37939611	14	13	29%
AVX2_E0	2991	119285	10616947	7	20937	30518927	18	21	42%
AVX2_E2	2619	121012	9672043	6	15714	22057913	21	25	37%

for the scalar ME kernel. However, with the vectorized ME, the performance is improved by only 10% to 15% on the Haswell system. Three factors can contribute to this limited improvement. First, the total number of memory accesses is greatly reduced by the vectorized computations (e.g., $\approx 10\times$ less number of memory accesses for SSE). Second, the copy-candidate sizes can also be affected by the length of the vector register and may increase the overhead of copying the data into the buffer. Finally, the relative stalled CPU cycles on the memory subsystems are increased due to the increased bandwidth demands of SIMD computations, which can be clearly seen in Figure 5.

Figure 4 and Table 4 present the multithreaded performance of different kernel implementations at different core frequencies. From the figure, we can observe that the performance of all of the kernels increases with the increasing number of threads as long as only one thread runs on a given physical-core. *AVX2_E2* provides the best performance in all cases and can

Table 4: Average execution time and core-energy consumption of multithreaded kernel implementations on Haswell system at a peak core frequency.

ME kernel	Execution Time				Core Energy			
	1-thread	2-threads	4-threads	8-threads	1-thread	2-threads	4-threads	8-threads
Scalar_E0	48480.38	25384.31	13714.04	19854.64	131.66	135.28	162.45	216.62
Scalar_E1	50745.10	26383.63	19607.79	18884.49	136.00	138.97	157.97	203.58
Scalar_E2	20491.06	11200.93	6394.28	12140.29	49.35	49.80	55.82	113.85
SSE_E0	5421.62	3547.64	2844.55	8445.98	12.60	12.02	13.67	60.22
SSE_E2	4697.03	3188.25	2395.11	7292.13	10.42	10.10	9.41	50.97
AVX2_E0	2991.48	1842.71	1535.45	6510.13	6.21	6.49	7.41	32.52
AVX2_E2	2619.03	1733.14	1502.45	6412.13	5.18	5.51	5.13	27.58

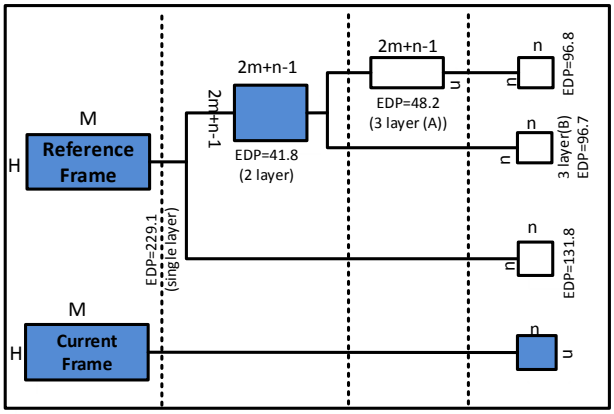


Figure 6: Energy-efficiency optimization using different possible data-reuse transformations on Sandy Bridge system.

provide speedup up to 34× on the Haswell system over the unoptimized scalar kernel (*SSE_E0*) when four threads are used. However, beyond four threads, when hyper-threading is used, the performance is degraded. This performance is less than what we could potentially achieve with the combination of multithreading (≈4×), vectorization (≈16× using SSE and ≈32× using AVX2 for an 8-bit value) and data reuse transformations.

In summary, based on our observation, we can conclude that vectorization accelerates the performance of the motion estimation kernel by reducing the total number of instructions and memory accesses through data parallelization. On top of that, data reuse transformations reduce expensive memory accesses to upper cache levels, by improving locality on lower levels. When combined with vectorization, data reuse transformation helps to keep the amount of stalled cycles due to memory accesses low, despite the extra pressure on the memory system. Finally, thread-level parallelism provides further performance improvements on multi-core platforms regardless of the frequency, as long as physical cores are used.

5.4 Energy efficiency analysis on the Haswell and Sandy Bridge Platforms

In this section, we present the implications of different strategies on improving the energy efficiency. Figure 6 presents the EDP measurements on the Sandy Bridge system for the different data reuse transformation options presented in Figure 2 for a sequential ME kernel.

The use of data reuse transformations in the motion estimation kernel re-

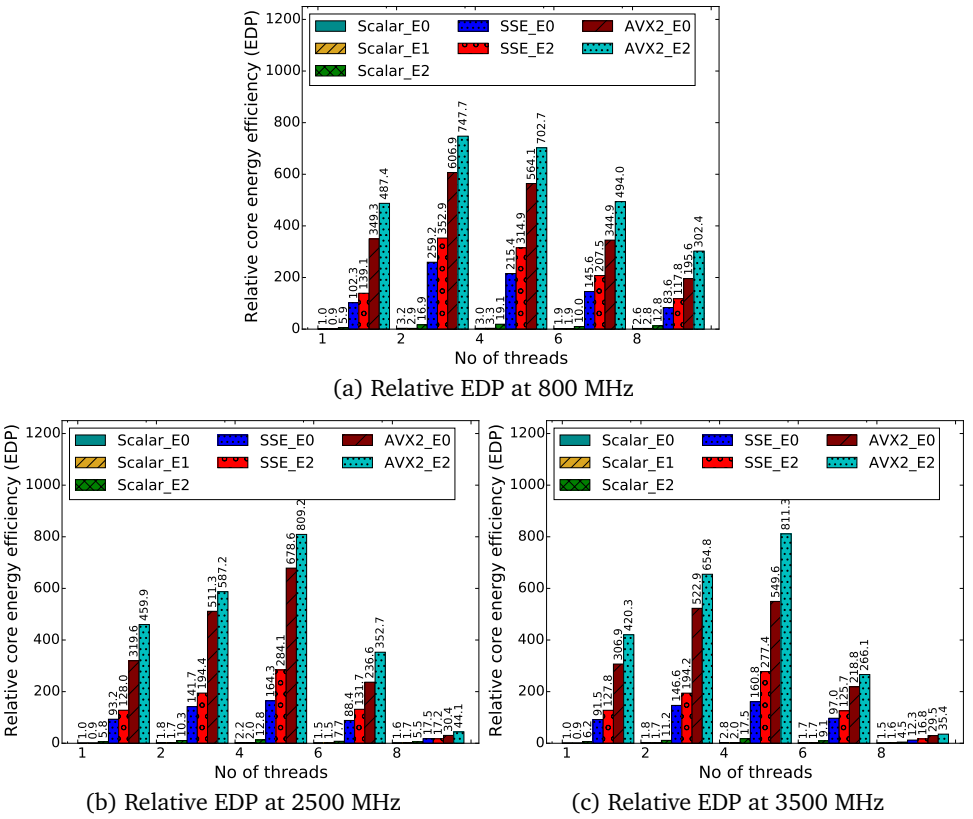
Table 5: Results of different data reuse transformations on Sandy Bridge system.

Version	**Execution Time** millisec	**Energy** milliJoule	**Energy Efficiency** (EDP) Js$\times 10^{-9}$	**Relative** Energy	**Relative** EDP
Scalar_E0	0.01379	378.71	5224.4	1.00	1.00
Scalar_E2	0.00461	130.29	608.8	2.91	8.58
Scalar_E3	0.00488	137.93	673.2	2.74	7.76
Parallel_E0	0.00143	141.03	201.1	2.69	25.98
Parallel_E2	0.00135	139.18	187.9	2.72	27.80

duces the accesses to the larger memories in the memory hierarchy. Therefore, it is expected that the kernels with data reuse transformations consume less energy than the unoptimized kernels. Indeed, Figure 6 shows a significant improvement in energy efficiency due to the data-reuse transformation techniques. The core energy consumption of the sequential motion estimation kernel is reduced to one-third of its original energy consumption by the deployment of data reuse transformations (in Table 5). In terms of the energy-delay product, the EDP of *Scalar_E0* is 5224.4 Js$\times 10^{-9}$, whereas the EDP of the *Scalar_E2* kernel that uses an additional memory hierarchy of block size $(2m + n - 1) \times (2m + n - 1)$ is 608.8 Js$\times 10^{-9}$, which is an $\approx 9\times$ improvement in energy efficiency in terms of EDP.

An important observation from Figure 6 is that the efficiency peaks with a two-layer memory hierarchy of $(2m+n-1) \times (2m+n-1)$ block memory and degrades with the introduction of any additional layers of smaller memory blocks. Two factors that can contribute to this result are: (i) data-reuse transformations generally make the code more complex and increase the code size; (ii) due to fixed-size caches, smaller data blocks are mapped to relatively larger cache blocks, which negate the advantage of using additional memory layers.

On the Haswell system, we also see that relative EDP measurements are improved by the data-reuse transformations techniques. Figure 7 illustrates that both parallelization and vectorization improve the energy efficiency in terms of EDP for the optimized (that exploits the data-reuse transformation methodology), as well as the unoptimized ME kernels (that does not use data-reuse transformations). Among the different possible ME kernels, *AVX2_E2* consumes the least amount of core energy (shown in Figure 3b), and the relative EDP values are improved rapidly with increasing number of threads and reach the maximum value when the total number of threads is four (in Figure 7). However, once the number of threads exceeds four,



(a) Relative EDP at 800 MHz

(b) Relative EDP at 2500 MHz

(c) Relative EDP at 3500 MHz

Figure 7: Relative EDP of multithreaded ME-kernel implementations at different core frequencies on the Haswell system. Higher relative EDP value indicates better energy efficiency.

relative EDP begins to decline.

It is also interesting to note that the multithreaded parallelization of the motion estimation kernel does not lead to core-energy savings despite the reduction in execution time due to parallelization at a peak core frequency on the Haswell system. This is an expected behavior if proper power saving mechanisms are in place, meaning that idle cores go into low power mode and do not contribute significantly to the total energy consumed during the application run. This is illustrated in Figure 3c. In contrast, vectorization results in quite a significant amount of core energy savings for the motion estimation kernel. This is due to several reasons. First, Intel processors do not have a separate vector unit or separate vector registers. This translates

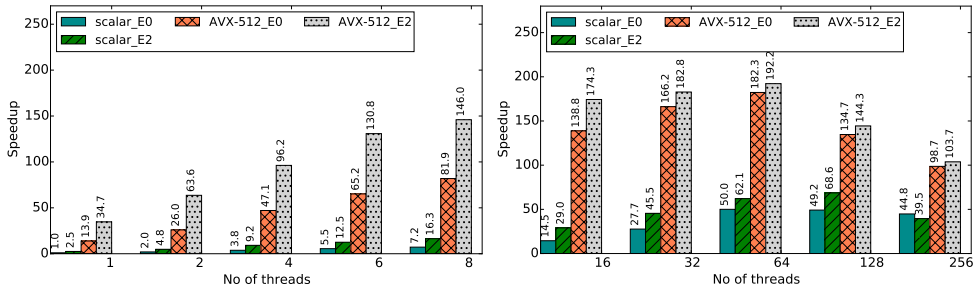


Figure 8: Achieved speedup of multithreaded ME-kernel implementation on KNL Xeon Phi processor.

into a small increase in the power dissipated by the ALUs/register bank working with SIMD instructions instead of scalar. In fact, GCC/ICC compilers no longer generate the scalar assembly, but rather SIMD instructions working only with the lowest vector lane. In addition, when working with vectors, the system spends more idle waiting time for memory, and therefore, cores can go into low power mode more often. This demonstrates that vectorization can lead to significant core energy savings if they can be applied effectively.

Table 5 gives a summary of our results for the Sandy Bridge platform. They show that data reuse transformations significantly improve the energy efficiency of the ME algorithm. The Relative EDP column in the table presents EDP values of different approaches normalized with respect to the EDP value of optimized parallel ME kernel. Relative EDP values indicate that the best energy efficiency can be achieved by using the parallel-optimized solution. Compared to the optimized serial solution (*Scalar_E2*), the parallel optimized solution (*Parallel_E2*) gives 3× better EDP, and the serial unoptimized solution (*Scalar_E0*) provides 28× higher EDP.

5.5 Performance and energy efficiency analysis on the KNL coprocessor

In our last set of experiments, we study the same problem in the many-core context. To this end, the best performing transformation options (*E2* kernels) are chosen along with the baseline (*E0*) to carry out scalability tests on the Intel Xeop Phi Co-processor (KNL). However, unlike the experiments done on the multi-core platforms, experiments on the KNL platform deal with full HD frames (1920×1080) as an input rather than the QCIF format. Figures 8 and 9 present improvements for the speedup and energy efficiency metrics that can be achieved through the optimized kernels when run on up to 256 threads on KNL (64 cores running four threads each; the

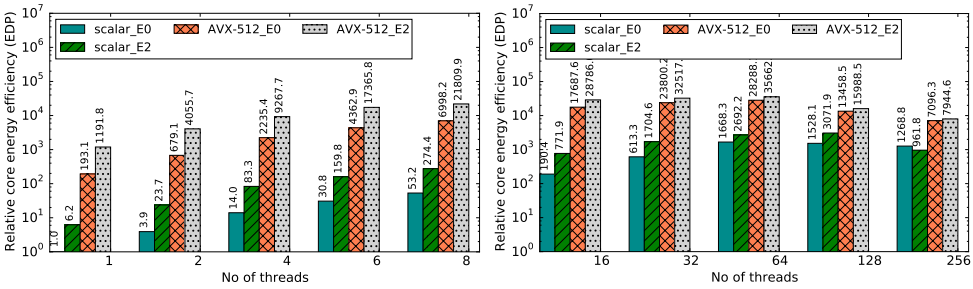


Figure 9: Relative EDP of multithreaded ME-kernel implementations on KNL Xeon Phi processor.

rest of the cores are reserved to manage the operating system). In Figure 8, observations on the single core performance highlight two interesting aspects of the kernels being investigated. First, *Scalar_E0* (i.e., unoptimized) and *Scalar_E2* (i.e., optimized using data reuse transformations) kernels exhibit $\approx 3\times$ performance increase, which is similar to what was observed on the Haswell system. Therefore, we can conclude that the data reuse transformations on a single core in a KNL coprocessor are as effective as the transformations on a single core in the Haswell system. Second, once again, the performance difference between a non-vectorized and vectorized kernel is quite significant; for example, the single-threaded *AVX512_E2*) kernel is $\approx 35\times$ faster than the single-threaded (*Scalar_E0*) kernel on the Haswell system (Figure 4c). Third, a significant amount of performance improvement can be observed by the deployment of data reuse transformations in conjunction with the AVX512 vectorization technique. Particularly for lower number of threads (i.e., <8), *AVX512_E2* provides $\approx 2\times$ better performance than *AVX512_E0*. Furthermore, we can also observe that the performance is increased with the increasing number of threads until it passes 64 threads and simultaneous multithreading takes place.

In terms of energy efficiency, we have also achieved improved results as shown in Figure 9. This metric is also scalable across multiple threads as long as only one thread runs per core. The energy efficiency improvements on the KNL platform are much larger than for the Haswell platform, which may seem surprising. The governor for our KNL system is set to performance, and it will therefore never throttle the core frequency down. In addition, there is probably not enough time to disable cores. According to Intel, it takes around one minute for an idle KNL core to go completely offline. On low core count, the OS may be jumping/sending system processes to different cores, so none will ever go offline. At best, the OS could use

DVFS (Dynamic voltage and frequency scaling) to reduce core frequency, but our governor prevents this. Since we do not have root access, we cannot test shield cores (prevent scheduling of OS processes into specific cores), nor bind all OS processes into a single core, nor change the governor. Nevertheless, this would be the common case for most end users.

5.6 Summary of Findings

We now summarize the key results and observations from our performance and energy efficiency evaluation of different data reuse transformations on Intel multi- and many-core systems.

1. Use of data reuse transformations together with vectorization is a promising approach to improve the performance and energy efficiency of massively parallel data-dominated applications (such as motion estimation) on multi- and many-core systems. Significant energy improvements can be achieved from throughput-oriented architectures that rely on low-power processing cores (e.g., KNL cores), especially if those cores provide SIMD/vector capabilities. These architectures have better energy efficiency (simple cores with low clock frequency) than complex cores available in commodity CPUs.

2. As compared to multi-threaded parallelism, data-parallelism through vector processing results in better energy savings even at peak core frequency. While doubling the number of cores results in approximately double the average power dissipated by the CPU, using vector units in Intel comes almost "for free" in terms of average power. Similar results have also been reported for several Intel and ARM CPUs in [2]. As a consequence, vector processing can be an attractive solution to improve energy efficiency without sacrificing performance, especially in a situation where performance trade-off is not desirable.

3. The deployment order of different optimization techniques has a great impact on the application performance. First, we apply multithreading to exploit explicit parallelism across multiple cores, which is followed by fine-grained parallelism through vectorization at each core. Finally, data reuse transformations are applied as it depends on both multithreading and vectorization for further improvement. However, on applications that face scalability issues, users may want to limit the amount of threads running in their application and rely more on SIMD units, since the energy cost of running on extra physical cores is much higher than using SIMD instructions.

4. In contrast to the results of Wuytack et al. in [5] where they have shown that a three-layer memory hierarchy is the most energy-efficient scheme for the ME algorithm, our results show that a two-layer memory hierarchy is more energy efficient. However, there exists a fundamental difference between these two experiments: First, we conduct our experiment on multi-core and many-core systems with a memory hierarchy of fixed sized cache-blocks, and thus, the copy-candidates are mapped into these fixed-size system caches. Since we cannot manually turn off the part of the caches not being used, our measurements include the energy consumed by both used and idle cache lines. Wuytack et al. avoided this extra energy consumption by conducting their experiment in a simulation environment where they created a hierarchy of memory blocks perfectly fitting the data blocks.

6 Conclusion

In this paper, we have investigated the performance and energy efficiency effects of applying data-reuse transformations on a multi-core processor running a motion estimation algorithm. We have shown that the performance can be improved up to 35×, and core energy consumption can be reduced by 25× on multi-core platforms (Haswell and Sandy Bridge) using appropriate data-reuse transformation techniques in combination with parallelization and vectorization. For a KNL many-core processor platform, this improvement can reach up to 192× and 185× (EDP 35,662×) for performance and core energy efficiency respectively when it runs with 64 threads. This gives clear indications that a data reuse methodology in combination with a parallel programming model can significantly save energy, as well as improve the performance of this type of application running on multi- and many-core processors.

In our experiments, simultaneous multithreading causes performance degradation. As our study was only limited to static and dynamic scheduling (in the KNL coprocessor), we plan to further extend our study to analyze the effect of using a more advanced scheduling method (e.g., guided scheduling) along with compiler-assisted selected lock assignment on the data reuse transformations in the simultaneous multithreading environment.

In the future, we will extend our study by running the experiments on an execution platform supporting a concept like drowsy cache [34] that powers down the unused parts of the cache. This would give more comparable results against the results of Wuytack et al.

Acknowledgments

The work presented in this paper was supported by the Faculty of Information Technology and Electrical Engineering, Norwegian University of Science and Technology.

Author contributions

All authors contributed extensively to the work presented in this paper. A.A.H. and P.G.K. developed the initial concept of the paper, and the extension of the initial concept was made by A.A.H. and L.N., and later on agreed upon by P.G.K.. A.A.H. designed and implemented the experiments under the supervision of L.N. and P.G.K. for multi-core systems and J.M.C. for the KNL coprocessor. A.A.H. and J.M.C. conducted the experiments on multi-core and many-core systems, respectively. All authors discussed the results and implications and commented on the manuscript at all stages.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] S. Ashby, P. Beckman and J. Chen. *The Opportunities and Challenges of Exascale Computing*. Report of the Advanced Scientific Computing Advisory Committee (ASCAC) subcommittee at the US Department of Energy Office of Science. 2010.

[2] Juan M. Cebrián, Magnus Jahre and Lasse Natvig. 'ParVec: Vectorizing the PARSEC Benchmark Suite'. In: *Computing* 97.11 (2015), pp. 1077–1100. ISSN: 1436-5057.

[3] Davendar Kumar Ojha and Geeta Sikka. 'A Study on Vectorization Methods for Multicore SIMD Architecture Provided by Compilers'. In: *Advances in Intelligent Systems and Computing* 248.1 (2014), pp. 723–728. ISSN: 978-3-319-03107-1.

[4] Changkyu Kim, Nadathur Satish, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar and Pradeep Dubey. *Technical Report: Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology*. 2012.

[5] J.Ph. Diguet, S. Wuytack, F. Catthoor et al. 'Formalized Methodology for Data Reuse Exploration for Low-Power Hierarchical Memory

Mappings'. In: *IEEE Transactions on VLSI Systems* 6 (1998), pp. 529–537.

[6] Francky Catthoor, Koen Danckaert, Chidamber Kulkarni, Erik Brockmeyer, Per Gunnar Kjeldsberg, Tanja Van Achteren and Thierry Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002. ISBN: 9780792376897.

[7] Francky Catthoor, Sven Wuytack, G.E. de Greef and et al. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792382889.

[8] Nikos D. Zervas, Kostas Masselos and C. E. Goutis. 'Data-Reuse Exploration for Low-Power Realization of Multimedia Applications on Embedded Cores'. In: *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation*. PATMOS'99. 1999, pp. 71–80.

[9] Alexander Chatzigeorgiou, Er Chatzigeorgiou, Stamatiki Kougia and et al. *Evaluating the Effect of Data-Reuse Transformations on Processor Power Consumption*. 2001. URL: `http://egnatia.ee.auth.gr/~alec/patmos2001.pdf`.

[10] N. Vassiliadis, A. Chormoviti, N. Kavvadias and et al. 'The Effect of Data-Reuse Transformations on Multimedia Applications for Application Specific Processors'. In: *Proceedings of the International Conference on Intelligent Data Acquisition and Advanced Computing Systems Technology and Applications*. IDAACS'05. Sept. 2005, pp. 179–182.

[11] Hari Kalva, Aleksandar Colic, Adriana Garcia and Borko Furht. 'Parallel Programming for Multimedia Applications'. In: *Multimedia Tools Applications* 51.2 (2011), pp. 801–818.

[12] Long Chen, Ziang Hu, Junmin Lin and et al. 'Optimizing the Fast Fourier Transform on a Multi-core Architectures'. In: *Proceedings of the Parallel and Distributed Processing Symposium*. IPDPS'07. Mar. 2007, pp. 1–8.

[13] Yuanrui Zhang, Mahmut Kandemir and Taylan Yemliha. 'Studying Inter-core Data Reuse in Multicores'. In: *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '11. 2011, pp. 25–36.

[14] G. He, D. Zhou, Y. Li, Z. Chen, T. Zhang and S. Goto. 'High-Throughput Power-Efficient VLSI Architecture of Fractional Motion Estimation for Ultra-HD HEVC Video Encoding'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.12 (Dec. 2015), pp. 3138–3142. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2386897.

[15] Jonathan Lifflander, Sriram Krishnamoorthy and Laxmikant V. Kale. 'Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 857–868. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.75.

[16] Paul Marchal, Francky Catthoor, Davide Bruni and et al. 'Integrated Task Scheduling and Data Assignment for SDRAMs in Dynamic Applications'. In: *IEEE Design & Test of Computers* 21.5 (Sept. 2004), pp. 378–387. DOI: 10.1109/MDT.2004.66.

[17] Abdullah Al Hasib, Per Gunnar Kjeldsberg and Lasse Natvig. 'Performance and Energy Efficiency Analysis of Data Reuse Transformation Methodology on Multicore Processor'. In: *Proceedings of the Euro-Par 2012: Parallel Processing Workshops*. Vol. 7640. LNCS. 2013, pp. 337–346. ISBN: 978-3-642-36949-0.

[18] James Jeffers James Reinders Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming*. Morgan Kaufmann Publishers Inc., 2016.

[19] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien and Ravi Nair. 'Data Access Optimization in a Processing-in-memory System'. In: *Proceedings of International Conference on Computing Frontiers*. CF '15. Ischia, Italy, 2015, 6:1–6:8. ISBN: 978-1-4503-3358-0. DOI: 10.1145/2742854.2742863.

[20] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov and J. Dongarra. 'A Step towards Energy Efficient Computing: Redesigning a Hydrodynamic Application on CPU-GPU'. In: *Proceedings of the International Symposium on Parallel and Distributed Processing*. May 2014, pp. 972–981. DOI: 10.1109/IPDPS.2014.103.

[21] J. S. Miguel, M. Badr and N. E. Jerger. 'Load Value Approximation'. In: *Proceedings of the International Symposium on Microarchitecture*. Dec. 2014, pp. 127–139. DOI: 10.1109/MICRO.2014.22.

[22] I Z. Reguly, Endre Lfffdfffdszlfffdfffd, Gihan R. Mudalige and Mike B. Giles. 'Vectorizing Unstructured Mesh Computations for Many-core Architectures'. In: *Concurrency and Computation: Practice and Experience* 28.2 (2016), pp. 557–577. ISSN: 1532-0634. DOI: 10.1002/cpe.3621.

[23] Daniel Dabbelt, Colin Schmidt, Eric Love, Howard Mao, Sagar Karandikar and Krste Asanovic. 'Vector Processors for Energy-Efficient Embedded Systems'. In: *Proceedings of the International Workshop on Many-core Embedded Systems*. MES '16. Seoul, Republic of Korea: ACM, 2016, pp. 10–16. ISBN: 978-1-4503-4262-9. DOI: 10.1145/2934495.2934497.

[24] A. Podobas, M. Brorsson and K. Faxen. 'A Performance Comparison of Some Recent Task-based Parallel Programming Models'. In: *Proceedings of the International Conference on High-Performance and Embedded Architectures and Compilers*. Pisa, Italy, Jan. 2010.

[25] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. July 2011. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[26] Juan M. Cebrián, Magnus Jahre and Lasse Natvig. 'Optimized Hardware for Suboptimal Software: The Case for SIMD-aware Benchmarks'. In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. Mar. 2014, pp. 66–75.

[27] Intel. *Intel 64 and IA-32 Architectures Software Development Manual*. Dec. 2011. URL: http://download.intel.com/products/processor/manual/325462.pdf.

[28] T. Komarek and P. Pirsch. 'Array Architectures for Block Matching Algorithms'. In: *IEEE Transactions on Circuits and Systems* 36.10 (Oct. 1989), pp. 1301–1308.

[29] J. Ott, C. Borman, G. Sullivan and et al. *RTP Payload Format for ITU-T Rec. H.263 Video*. 4629. Jan. 2007. URL: http://tools.ietf.org/html/rfc4629.

[30] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan and et al. 'Models and Metrics to Enable Energy-Efficiency Optimizations'. In: *Computer* 40.12 (Dec. 2007), pp. 39–48.

[31] Hallgeir Lien, Lasse Natvig, Abdullah Al Hasib and Jan Christian Meyer. 'Case Studies of Multi-core Energy Efficiency in Task Based Programs'. In: *Proceedings of the International Conference on ICT as Key Technology against Global Warming*. Vol. 7453. Sept. 2012, pp. 44–54.

[32] *Performance Application Programming Interface*. URL: `http://icl.cs.utk.edu/papi/index.html`.

[33] John Russell. *ARM Unveils Scalable Vector Extension for HPC at Hot Chips*. URL: `https://www.hpcwire.com/2016/08/22/arm-unveils-scalable-vector-extension-hpc-hot-chips/`.

[34] Krisztián Flautner, Nam Sung Kim, Steve Martin and et al. 'Drowsy Caches: Simple Techniques for Reducing Leakage Power'. In: *Proceedings of the Annual International Symposium on Computer Architecture*. ISCA '02. Washington, DC, USA, 2002, pp. 148–157.

Paper C.2

A Vectorized *K-means* Algorithm for Compressed Datasets – Design and Experimental Analysis

Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig

Journal of Supercomputing, 2018

Abstract

Clustering algorithms (i.e., gaussian mixture models, *k-means*, etc.) tackle the problem of grouping a set of elements in such a way that elements from the same group (or cluster) have more similar properties to each other than to those elements in other clusters. This simple concept turns out to be the basis in complex algorithms from many application areas, including sequence analysis and genotyping in bio-informatics, medical imaging, antimicrobial activity, market research, social networking etc. However, as the data volume continues to increase, the performance of clustering algorithms is heavily influenced by the memory subsystem.

In this paper, we propose a novel and efficient implementation of Lloyd's k-means clustering algorithm to substantially reduce data movement along the memory hierarchy. Our contributions are based on the fact that the vast majority of processors are equipped with powerful Single Instruction Multiple Data (SIMD) instructions that are, in most cases, underused. SIMD improves the CPU computational power and, if used wisely, can be seen as an opportunity to improve on the application data transfers by compressing/decompressing the data, specially for memory-bound applications. Our contributions include a SIMD-friendly data-layout organization, in-register implementation of key functions and SIMD-based compression. We demonstrate that using our optimized SIMD-based compression method, it is possible to improve the performance and energy of k-means by a factor of $\approx 5x$ and $\approx 9x$ respectively for a i7 Haswell machine, and $\approx 22x$ and $\approx 22x$ for Xeon Phi: KNL, running a single thread.

1 Introduction

Clustering algorithms try to group a set of elements in such a way that elements from the same group (or cluster) have more similar properties to each other than to those elements in other clusters. Clustering is considered as a central problem in data management and data mining, as well as the basis in complex algorithms from many fields of application. Clustering is used in bio-informatics for sequence analysis and genotyping, to group homologous sequences into gene families. On PET[1] scans (medical imaging), cluster analysis can be used to differentiate between different types of tissue and blood in a three-dimensional image. It can also be used to analyse patterns of antibiotic resistance in medical research, to analyze multivariate data from surveys and test panels in market research or to to recognize communities within large groups of people in social networks.

Among the many different clustering methods, *k-means* is one of the most widely used. The advantage of *k-means* is its simplicity: starting with a set of randomly chosen initial centers, the kernel repeatedly assigns each input point to its nearest center, and then recomputes the centers given the point assignment. From a theoretical standpoint, *k-means* is not a good clustering algorithm in terms of efficiency or quality: the running time can be exponential in the worst case and even though the final solution is locally optimal, it can be far from the global optimum (even under repeated random initializations). Therefore, recent works e.g. *k-means++* focus on improving the initialization procedure, increasing performance, convergence and quality [1].

In recent years, we have witnessed an explosive growth of big data [2]. The overwhelming data inputs raise compelling computational challenges to data intensive kernels, such as clustering. Despite the advent of multi-core and many-core systems, the performance of data intensive computations is often largely inhibited by slow disk accesses as well as the limited bandwidth or latency for data transfers across the memory hierarchy. This is especially critical in real-time or near real-time scenarios (e.g., analyzing a high resolution medical image in a few hours rather than days can be extremely beneficial for a patient). Effective data compression algorithms can be used to mitigate this problem by reducing the amount of data to be transferred across the memory hierarchy as well as the number of required memory/disk accesses.

[1] Positron Emission Tomography.

Modern processors equipped with extra-wide registers for SIMD (Single Instruction Multiple Data) instructions provide us with an opportunity to achieve better compression performance. For instance, Intel has been supporting data parallelism through 128-bit and 256-bit SIMD computations using SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instructions and recently extends their support further by introducing AVX-512 (512-bit extensions to the 256-bit AVX) SIMD instructions. ARM is going to release Scalable Vector Extension (SVE) [3] instruction set to support up to 2048-bit vectors. Therefore, the emerging trends clearly show that vectorization is going to play an important role in the future High Performance Computing (HPC) systems. Consequently, many researchers seek to exploit the vector units in the recent hardware to improve the decoding speed of compression algorithms [4, 5]. In particular, the authors in [6] have demonstrated the effectiveness of using a SIMD compression method to improve performance and energy efficiency of cache/memory bound time series processing.

Based on the aforementioned findings, here we present a simple, yet efficient implementation of Lloyd's algorithm [7] by using an effective SIMD based compression approach to accelerate the performance of integer *k-means* clustering on compressed data sets. The idea is to improve data locality and decrease the memory bandwidth requirements of SIMD based computations by using a lightweight compression method without significantly increasing the computational requirements. The algorithm begins by storing data points in a *block data layout* format where each block is compressed using the V-PFORDelta coding method described in [6].

Our key contributions in this paper are:

- We make an efficient implementation of a state-of-the-art *k-means* algorithm (*Lloyd et al.* [7]) by optimizing its loop traversal scheme and by using a *block data layout* format to store the data. This is key to make the overall computations more SIMD-efficient and to improve the data locality of the algorithm. This layout also enables SIMD software prefetching to further improve performance.
- We introduce an in-register implementation of the most time-consuming function (namely *ArgMin*, discussed in Section 3) to optimize data locality and conserve memory bandwidth.
- The distance vector of the clustering algorithm does not need to be completely accurate, as long as elements are clustered in the same

way. We use a *scalar product* based approximation to the euclidean distance computation in order to reduce the computational requirements (Section 4.2).

- We present a method to reduce the increased pressure on the memory subsystem due to vectorization using a lightweight SIMD-based data compression method [6]. The underlying concept here is to reduce the total number of required memory accesses by accessing compressed data. We show that integration of compression is feasible, specially when the processor runs out of reservation stations or load-store-queue entries and memory-level parallelism (MLP) is degraded.

- Finally, we demonstrate the effectiveness of our proposed approach in terms of performance and energy-efficiency on Intel multi-core (Haswell) and many-core (Xeon Phi: Knights Landing (KNL)) platforms.

2 Related Work

Clustering problems have been frequent and important objects of study for the past many years by data management and data mining researchers. One of the most popular heuristics for solving these problems is based on a simple iterative scheme for finding a locally minimal solution, often known as *k-means* algorithm. There are a number of variants to this algorithm, so, to clarify which version we are using, we will set our baseline as Lloyd's algorithm [7]. The initialization process of the algorithm is crucial for obtaining a good solution [8].

In this paper we primarily focus on the acceleration of the *k-means* algorithm using thread and data-level parallelism. This goal has been the target of many recent researches to accelerate the *k-means* algorithm. In [9], the authors have explored the performance of general-purpose applications including a CUDA implementation of a *k-means* algorithm on graphics processors. In [2, 10], the authors proposed an algorithm that performs the distance calculations in parallel on the GPU while sequentially updating the cluster centroids on the CPU based on the results from the GPU calculations. In the aforementioned optimization methods, parallelism is done at the task level, where the data is divided into smaller chunks and each chunk is processed in sub-tasks. All of these tasks execute the same logic as in the baseline *k-means* algorithm. In contrast, our proposed implementation method slightly differs from the baseline as it integrates a compression method for further optimization.

In [11], Hadian and Shahrivari have used a KD-tree (k-dimensional tree) based structure where each node for the KD-tree is represented by a bounding box specifying the minimal axis-parallel hyper-rectangle containing all associated points. Consequently, the search for nearest centroid is accelerated. Our work is closely related to the paper [12], where the authors proposed a fine-grained SIMD based approach which computes n distances from the n data points to the same centroid in one loop. Hence, this approach is termed as centroid-oriented approach. We have made an improvement over this approach by performing in-register *Arg-min* computation along with computations using compressed data set. Moreover, none of the aforementioned approaches has performed the energy efficiency analysis. In addition, to the best of our knowledge, the systematic investigation of *k-means* implementation using SIMD instruction sets has not been performed on Intel's Knights Landing platform before.

3 K-means Clustering Overview

3.1 Single-threaded Scalar K-means

Let $X = \{x_1, \ldots, x_n\}$ be a set of data points in the d-dimensional space and let k be a positive integer specifying the number of clusters. Let $C = \{c_1, \ldots, c_k\}$ divide X into k clusters ($X'_j \subset X$, $j = \{1, \ldots, k\}$), where c_j is the centroid of cluster X'_j. The distance from a data point (x_j) to a centroid (c_j) is determined by the Euclidean Distance denoted as $\phi(x_i, c_j) = \sqrt{\sum_{k=1}^{d}(x_i^k - c_j^k)^2}$. The optimal set C of k centroids can be found by minimizing the following function:

$$\Theta = \sum_{x_i \in X, c_j \in C} \phi(x_i, c_j)$$

A single-threaded scalar *k-means* algorithm is illustrated in Algorithm 1. The algorithm is divided into 3 states namely seeding state, labeling state and cluster update state. In the *seeding state*, the initial set of centroids is chosen by k random values from the set of data points X. In every iteration of the *labeling state*, each data point $x_i \in X$ is assigned to the cluster C_{l_i} with the closest centroid c_{l_i}. This is implemented in the $\underset{c_j \in C}{\mathrm{ArgMin}}\ \phi(x_i, c_j)$ function, which returns the number l_i (the label for datapoint x_i) of the cluster that minimizes $\phi(x_i, c_j)$ in line 5. Line 6 forms the new updated clusters, but does not update the position of its centroid, which is done in next state, the *cluster update state*. Here the new cluster value (the position

Algorithm 1 Sequential *k-means* Algorithm
Input: data (X), number of clusters (k) | Output: centroids (C)

Seeding state

1: $c_j \in C \leftarrow$ random $x_i \in X$, $i = 1..., n$; $j = 1, ..., k$

Labeling state

2: **repeat**
3: $X'_{1..k} = \{\}$
4: **for** Each $x_i \in X$ **do**
5: $l_i \leftarrow \underset{c_j \in C}{\mathrm{ArgMin}}\ \phi(x_i, c_j)$
6: $X'_{l_i} \leftarrow X'_{l_i} \cup x_i$
7: **end for**

Cluster update state

8: **for** Each $c_j \in C$ **do**
9: $c_j \leftarrow \frac{1}{|\ X'_j\ |} \sum x_j \in X'_j$
10: **end for**
11: **until** convergence

of the centroid, c_j) is computed for each of the updated clusters X'_j by dividing the sum of the cluster-values with the number of data points in each cluster (namely m_j).

3.2 Multi-threaded Scalar K-means

We use the OpenMP [13] API to make a parallel implementation of the algorithm. In the parallel implementation of the *k-means* algorithm, the labeling state (i.e. l_i computation) is identified as being inherently data parallel. Therefore, Algorithm 1 can be translated into a multi-threaded implementation by the following two steps (See Algorithm 2):

- Divide the data set X to be clustered into *p* blocks and assign one thread for each block. Each thread executes independently the labeling step (lines 5–9) in parallel. This implies also to update its partial *sum* of points in cluster l_i and m-value (number of elements in cluster).
- Once the labeling step is completed, an implicit barrier allows the reduction step to combine all thread-local partial *sum*- and *m*-values together [13]. Then we update the centroids accordingly.

Algorithm 2 Parallel *k-means* Algorithm
Input: data (X), number of clusters (k) | Output: centroids (C)

1: $c_j \in C \leftarrow$ random $x_i \in X$, j=1, ..., k
2: **repeat**
3: $sum_{1..k} = m_{1..k} = 0$
4: #pragma omp parallel for reduction(+: $sum_{1..k}, m_{1..k}$)
5: **for** Each $x_i \in$ my block of X **do**
6: $l_i \leftarrow \underset{c_j \in C}{ArgMin}\ \phi(x_i, c_j)$
7: $sum_{l_i} \leftarrow sum_{l_i} + x_i$
8: $m_{l_i} \leftarrow m_{l_i} + 1$
9: **end for**
10: **for** Each $c_j \in C$ **do**
11: $c_j \leftarrow \frac{sum_j}{m_j}$
12: **end for**
13: **until** convergence

4 Multi-threaded Vectorized K-means with Compressed Dataset

To exploit the full performance potential of the modern micro-architectures supporting SIMD operations, we have carefully chosen the following strategies:

- Memory hierarchy sensitive strategies to efficiently transfer data into the registers.
- Approximate the Euclidean distance computation by using precomputed *scalar products*.
- In-register *ArgMin* computation to optimize data locality.
- SIMD data compression to conserve bus-bandwidth and reduce the number of cache accesses.

4.1 Loop Traversal Strategy and Data Layout Optimization

The *ArgMin* computation step of the *k-means* algorithm is typically implemented by three nested loops iterating over data points (n), cluster representations (k) and dimensions (d). Therefore, the loop traversal strategy of this step can also be defined by the *n-k-d* space. In order to optimize the data locality property of this *ArgMin* computation, we divide the input data stream into smaller blocks.

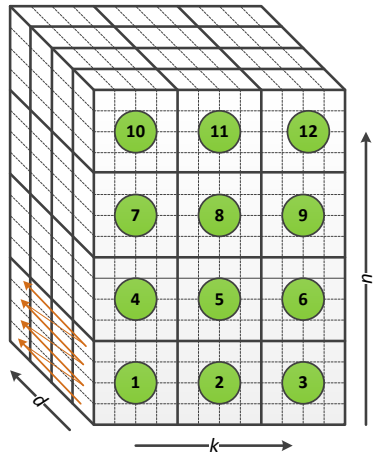


Figure 1: Loop traversal strategy for SSE-SIMD *k-means*.

Figure 1 illustrates the loop traversal strategy that is deployed in our multi-threaded vectorized *k-means* implementation. In the figure we can observe that the outermost loop iterates over the data points x_i, the nested loop iterates over the cluster representatives c_j and the innermost loop iterates over the dimensions d. For SSE, each block in the *n-k-d* space involves 4 data points (the same as the number of uint32_t values we can fit in a SSE register), 4 cluster representatives (also equal to the data we can fit in the SSE register) and 1 dimension. Once this step is repeated over the d dimension, as indicated by the orange zig-zag arrow in the Figure 1, the *ArgMin* computations for the 4 data points (green circles) are completed. Next, the *ArgMin* computations for the next 4 points can begin.

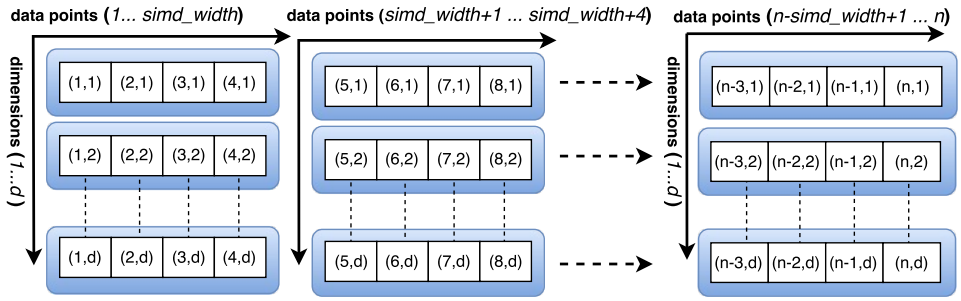


Figure 2: Block data layout for locality optimization.

It is important to note that all the data points fitted into the vector registers first complete their computations across all dimensions (d), before the next set of data points are loaded into the vector register. Hence, neither the row-major layout nor the column-major layout is efficient for storing the data block into the memory. Instead, a block data layout of ($s \times d$) is used to store the data into memory, where s is the vector register size divided by element size, or *SIMD_WIDTH*, and d is the dimension. Figure 2 illustrates the block data layout format used in our multi-threaded *k-means* implementation. The combination of block data layout with our chosen data access strategy has the following advantages:

- The loop traversal strategy is efficient as the n data points are streamed into the SIMD register only once. Also the k cluster representatives are streamed into the registers only once for the n different data points within a *n-k-d* block. As a result, it reduces the required number of repeated transfers of the cluster representatives to the registers.

- The block data layout minimizes memory bank conflicts by grouping the contiguously used data together. Also the disturbance of temporal data held within processors caches is minimized by using streaming store instructions[2].

- The SIMD based *ArgMin* computation does not require horizontal addition[3] anymore. Thereby the overall computations become more SIMD efficient.

4.2 Approximate Euclidean Distance Computation

In the *k-means* algorithm, the Euclidean distance metric is used for comparison purposes only, rather than computing the actual distance. Therefore, the distance vector does not need to be completely accurate. Considering this, we have adopted an indirect approach to compute this distance using a scalar product.

Let us consider, $\vec{x}_1$ and $\vec{c}_1$ represent a multi-dimensional data point and a cluster representative respectively, and the dimension is d in both cases. Let us also assume that $\{x_{1_1}, \ldots, x_{1_d}\}$ and $\{c_{1_1}, \ldots, c_{1_d}\}$ are the values of $\vec{x}_1$ and $\vec{c}_1$ across d dimensions. Now, the formulas for the Euclidean distance

[2] An store instructions that skips the first level of the cache hierarchy.

[3] The addition of all the data values within a vector register.

$\| \vec{x}_1 - \vec{c}_1 \|$ and scalar distance $\langle \vec{x}_1, \vec{c}_1 \rangle$ [14] can be defined as:

$$\| \vec{x}_1 - \vec{c}_1 \| = \sqrt{\sum_{i=1}^{d} (x_{1_i} - c_{1_i})^2}, \qquad \langle \vec{x}_1, \vec{c}_1 \rangle = \sum_{i=1}^{d} x_{1_i} \cdot c_{1_i}$$

The Euclidean distance computation can be re-written as:

$$\begin{aligned} \| \vec{x}_1 - \vec{c}_1 \|^2 &= \sum_{i=1}^{d} (x_{1_i} - c_{1_i})^2 \\ &= \sum_{i=1}^{d} (x_{1_i}^2 + c_{1_i}^2 - 2 x_{1_i} c_{1_i}) \\ &= \sum_{i=1}^{d} x_{1_i} \cdot x_{1_i} + \sum_{i=1}^{d} c_{1_i} \cdot c_{1_i} - 2 \sum_{i=1}^{d} x_{1_i} \cdot c_{1_i} \\ &= \langle \vec{x}_1, \vec{x}_1 \rangle + \langle \vec{c}_1, \vec{c}_1 \rangle - 2 \langle \vec{x}_1, \vec{c}_1 \rangle \end{aligned}$$

In the labeling state of *k-means* algorithm, the Euclidean distance between a data point $\vec{x}_1$ and all cluster representatives $\vec{c}_1$,...,$\vec{c}_k$ is computed. Therefore, we can pre-compute $\langle \vec{c}_1, \vec{c}_1 \rangle$, ..., $\langle \vec{c}_k, \vec{c}_k \rangle$ before starting the labeling state. As a result, the membership id (label) (l) of a data point $\vec{x}_i$ can be defined as:

$$\begin{aligned} l_i &= \underset{1 \le j \le k}{ArgMin} \ \| \vec{x}_i - \vec{c}_j \|^2 \\ &= \underset{1 \le j \le k}{ArgMin} \ \langle \vec{x}_i, \vec{x}_i \rangle + \langle \vec{c}_j, \vec{c}_j \rangle - 2 \langle \vec{x}_i, \vec{c}_j \rangle \\ &= \underset{1 \le j \le k}{ArgMin} \ \frac{1}{2} \langle \vec{c}_j, \vec{c}_j \rangle - \langle \vec{x}_i, \vec{c}_j \rangle \end{aligned}$$

since x_i is identical for all j, we can skip $\langle \vec{x}_i, \vec{x}_i \rangle$ computation and divide the operand of *ArgMin* by the positive constant two. Consequently this approximation computation requires *d* multiplications and *d* additions or subtractions and one array lookup as compared to original *d* multiplications and *2d-1* additions or subtractions.

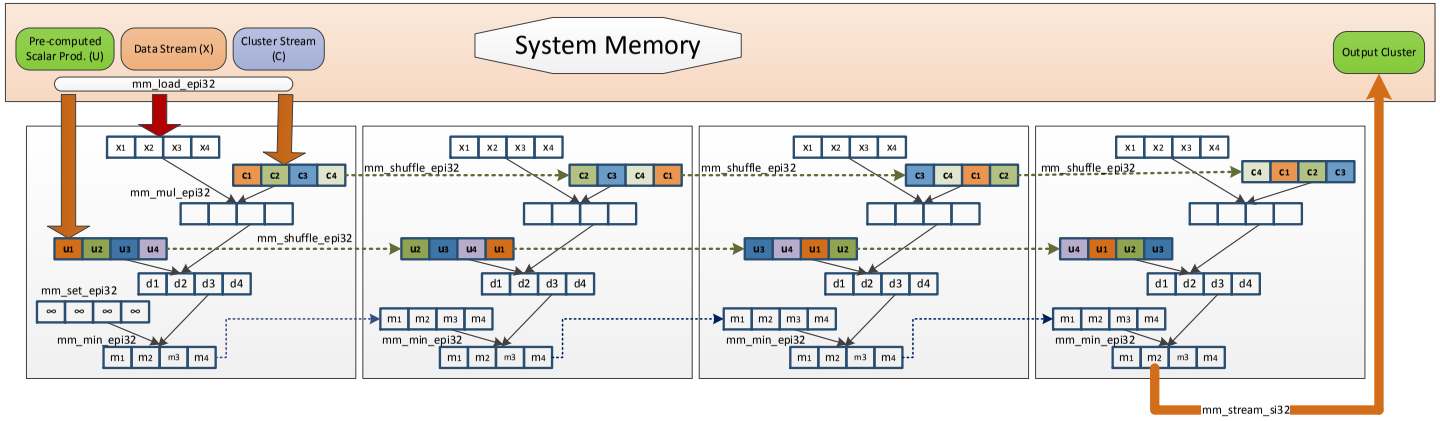


Figure 3: Work flow of in-register *ArgMin* call for 4 data points of d dimensions with 4 centroid values using SSE-SIMD intrinsics. Once the centroids values (i.e. c_1, c_2, c_3, c_4) are loaded into a SIMD register, the centroids are shuffled around in the distance computation step. This allows to compute four partial distances (i.e. $d_1^1, d_1^2, d_1^3, d_1^4$) between data points and centroids (c_1, c_2, c_3, c_4) for each of the loaded data points (e.g. x_1) using only a single load operation for the four centroid values.

4.3 In-register ArgMin Computation

As loading causes an entire cache line to be moved into the cache hierarchy, any load operation looks more or less the same from a memory bandwidth perspective irrespective of the size of the data operand. Moreover, many small loads often consume more microarchitectural resources, which may cause the processor to stall and reduce the MLP. Therefore, while performing *ArgMin* computation, we aim to minimize the required number of load operations while maximizing the utilization of the data that are loaded into the register. In our *ArgMin* computation scheme, as illustrated in Figure 3, the number of load operations is reduced as the block data layout puts relevant data close to each other. Additionally, we can reuse the loaded data in several iterations of the *ArgMin* computation by shuffling the register contents. For instance, once centroids values (i.e. c_1, c_2, c_3, c_4) are loaded into a SIMD register, the values are shuffled around to compute the distances of the 4-centroids from each data point (e.g. x_1). As a consequence, repetitive transfer of cluster representatives is avoided for each point x_i, thus reducing the memory bandwidth requirements.

4.4 Bus-bandwidth Conservation Through Vectorized Data Compression

We further attempt to reduce pressure on the memory subsystem by loading and storing more data to/from the same DRAM page. With this aim, we split the data stream into blocks of ($s \times d$) integers, where each block of data is compressed using a SIMD compression method. The process is similar to algorithm 2, but has an additional step where each block of data is decompressed in parallel (between lines 4 and 5).

The scope of the paper is limited to integer compression on integer inputs. Centroids are approximated to their nearest integer so that integer compression techniques can be applied. This is not mandatory in our proposal, but not doing so reduces the compression ratio for large high-dimensional cluster sets. This approximation is applied to all kernels in order to avoid reporting any unfair benefits that it may cause.

For compressing a data block we use the V-PFORDelta coding scheme proposed in [6]. V-PFORDelta is a delta coding-based compression technique which uses vectorized binary packing over blocks of integers. This scheme uses *b* bits to represent each integer value and stores exceptions that cannot be represented by *b* bits on a per block basis. Then, successive values are stored using *b* bits per integer using a fast bit packing functions. The factors that determine the storage cost of a given block in binary packing are:

- the number of bits (b) used to present the delta value.
- the block length ($B = s \times d$)
- a fixed per-block overhead (κ)

The total storage cost for one block is $bB + \kappa$. We tune the bit width (b) of delta to 16 to minimize the value of $((s \times d) \times b + c(w) \times 32)$ where $(s \times d)$ is the length of block and $c(w)$ is the number of exceptions. For further details about the SIMD implementation of V-PFORDelta please refer to [6].

5 Experiments and Results

There are many optimizations available for *k-means*, so it is hard to choose a baseline for comparison in our specific evaluation environment. In addition, replication of results if no source-codes are provided is a real challenge. To minimize the sources of error, we chose a simple algorithm [2] and the available OpenMP implementation as baseline. This selection ensures that we can isolate the effects of SIMD-friendly data structures and SIMD-compression from other optimizations [15].

Most of the related work optimizations are orthogonal to ours, and many others can be suitable for compression. Note that the main goal of this paper is "to improve on the behaviour of memory/latency bound applications through compression techniques". We are not trying to compete for best speedup, but to show the feasibility and what results can be expected from SIMD-based compression. Table 1 shows the expected compatibility with other optimizations available in the literature to achieve best performance.

Moreover, we are considering the following assumptions in our evaluation:

- Compression is done offline. In many big-data applications, specially those with low insertion count, storing datasets in compressed formats that can be directly accessed is the most promising solution.
- Centroids are approximated to their nearest integers. This is not an obligatory part of our proposal. The approximation can be avoided by one additional SIMD-conversion (int to float) on top of D (= SIMD-width) uncompressed integers to continue with floating point operations. This will prevent compressing the cluster values with the selected integer compression technique though. Also note that accuracy is not an issue, since iterative algorithms usually stop on a convergence

Table 1: Compatibility analysis of stat-of-the-art proposals with our proposed scheme

Paper	Contributions		Compatibility			
	Proposal	SIMD comp.	SIMD compression	Block-data-layout + loop-opt.	In-register distance comp.	Argmin Approx.
[1]	Improved seeding algorithm	No	Yes	Yes	Yes	Yes
[2]	Heterogeneous computation: labeling on GPU, cluster update on CPU	Yes	Yes	Yes	Yes	Yes
[16]	Use of KD-tree to filter out a candidate	No	Yes	No	No	Yes
[12]	Heterogeneous computation: centroid labeling on KNC, cluster update on CPU	Yes	Yes	No	No	Yes
[17]	Avoids distance computations using distance bounds and triangular inequality	No	Yes	No	No	Yes
[18]	Use of MapReduce, iteration dependence is reduced using probability sampling	No	Yes	Yes	Yes	Yes
[19]	Approximation using binary-tree cluster closure	No	Yes	No	No	Yes
[20]	Encode high dimensional data points	No	Yes	Yes	Yes	Yes

criteria, that is respected when using approximation to integers. The overhead in iteration count is relatively small ($< 4\%$).

5.1 Experimental Setup

We present the following seven variants of *k-means* implementations to demonstrate the effectiveness of our proposed approach:

- *Scalar*: A simple implementation of *k-means* algorithm using C++.
- *SSE_auto*: Auto-vectorization of *Scalar* implementation (using -msse2 compiler flag to prevent AVX code generation).
- *SSE_basic*: Hand-tuned SSE-based vectorization of *k-means* algorithm over data dimension.
- *SSE_optimized*: SSE-based vectorization of the proposed SIMD-optimized *k-means* algorithm.
- *SSE_compressed*: *SSE_optimized* implementation integrated with V-PFORDelta coding technique.
- *AVX512_auto*: Auto-vectorization of *Scalar* kernel using -xMIC-AVX512 flag.
- *AVX512_compressed*: Hand-tuned AVX512-based vectorization of the optimized kernel integrated with V-PFORDelta.

Table 2: Hardware Specifications of the Test Platforms

Processor	**Intel® Core™ i7-4700K**	**Intel® Xeon Phi 7250**
Architecture	Haswell	Knights Landing
Clock Speed	0.8 – 3.5 GHz	1.4 Ghz
# of Cores	4 cores / 8 threads	68 cores / 272 threads
L1 Cache	32 KB data + 32 KB inst, 8-way private	
L2 Cache	256 KB, 8-way private	1 MB, 16-way per 2 cores
L3 Cache	8 MB, shared, 16-way associativity	16 GB, shared HBM-MCDRAM

In this experiment, we have used Intel® Core™ i7-4700K desktop processing system. The system runs with Ubuntu 14.04.1 LTS 64-bit OS. Intel C++ compiler (version 14.0.1) with -O3 optimization flag is used to generate the executables. Turbo Boost Technology is disabled in the BIOS and the CPU frequency is set to a certain value while taking the measurements. In addition, we have also tested our implementations in a Xeon Phi 7250 processor with 68 cores running at 1.40 GHz. The system runs SUSE Linux Enterprise Server 12 SP1 and the binaries are generated using Intel C++ compiler (version 17.0.035) with -O3 optimization level and the -xMIC-AVX512 flag to generate AVX512 code. We only use 64 cores, and leave 4 cores to handle the OS (recommended by Intel). The hardware specifications of our test platforms are presented in Table 2.

Knights Landing (KNL) offers a high number of cores (68 in our evaluation platform) with up to four threads per core. The cores are based on *Silvermont Atom* out of order cores, tiled in pairs. Each core contains two Vector Processing Units (VPUs), that work with vector registers up to 512-bit wide. The VPUs are compatible with SSE, AVX and AVX512, but only one of the VPUs will be used for SSE-AVX codes. If the user wants to get the full potential of the VPUs the code needs to be recompiled for AVX512 (we recompiled SSE versions to run on KNL). In addition, each tile shares 1 MB of L2 memory, that are linked together using a 2D mesh interconnect (or NOC[4]). This interconnect hooks the cores to two DDR4 memory controllers (384 GB with a bandwidth of 90 GB/sec) and eight stacks of high bandwidth memory (HBM-MCDRAM, 16 GB with a bandwidth close to 400 GB/sec). The HBM memory can work in different modes, as a scratchpad memory, as an additional cache level or in hybrid mode (combination of the previous two modes). Our system is configured to use the HBM as cache (L3). Another key feature of KNL as compared to KNC or other Many-core platforms

[4]Network on Chip.

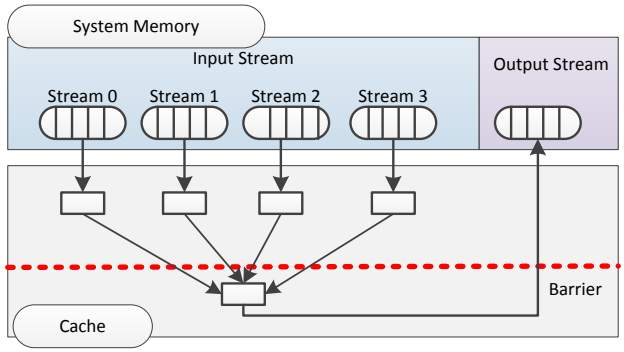


Figure 4: The basic workflow of parallel *k-means* using compressed data set on multi-core systems.

(like GPGPUs) is that it can work as an stand-alone processor, being the first bootable implementation of what was, up until now, a coprocessor handled by a host CPU.

5.2 Multi and Many-core Implementations

We have used OpenMP to achieve thread-level parallelism for our *k-means* algorithm. We implemented our code using a wrapper library, rather than writing intrinsics directly on the code. This wrapper library is contained in a header file that is imported by the *k-means* code, making the code more readable and easy to modify/migrated between architectures. For example, an integer SIMD addition (*simd_add_i(x,y)*) is defined in the wrapper library as a macro that translates to *_mm512_add_epi32(x,y)* for AVX512 and *_mm_add_epi32(x,y)* for SSE. The source code uses *simd_add_i(x,y)* for SIMD integer additions, and the pre-processor translates the macros to the appropriate target architecture. This implementation allows us to keep almost the same code for SSE and AVX512, except for instructions that merge register types (e.g., *cvtepi8_epi32*). Performance and energy of intermediate vector size implementations (i.e., AVX2 256-bit) are not shown to improve legibility but can be extrapolated from SSE (128-bit) and AVX512 (512-bit). The thread schedule is set as static (i.e. default) for the Haswell system, so that the iterations are partitioned into chunks which are allocated to the threads in a round-robin manner. The thread affinity is set as scatter to make the best use of each core first. The work flow of our proposed method is illustrated on Figure 4. However, the KNL system showed slightly better performance with the dynamic thread scheduling than static (around 5%) when working with high number of threads, even though we don't have

explicit synchronization between threads. Scatter thread affinity outperformed compact by a factor of 2*x* on both static and dynamic scheduling. The scheduling analysis is not shown since we feel it is not relevant to the publication, but it can be included upon request.

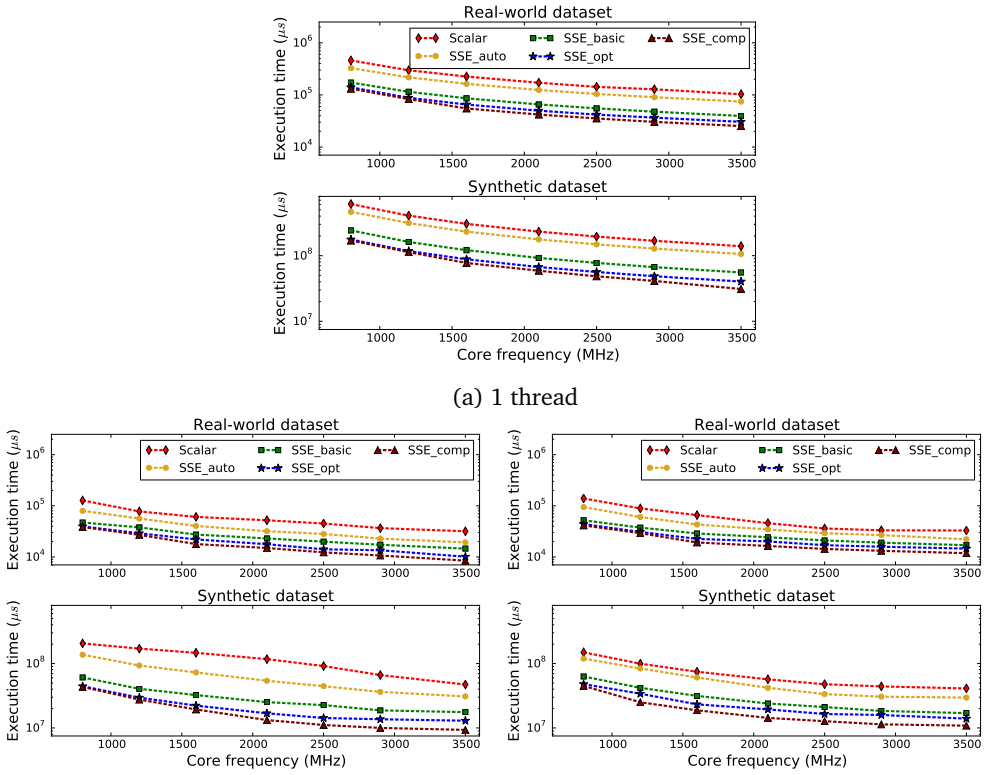
5.3 Datasets

Real-world Data

To understand the relative efficiency of this algorithm under practical circumstances, we use KDDCupBio04: a multidimensional biological dataset which is used in several scientific research works [21, 22, 23] involving clustering of high dimensional data. This dataset consists of 145751 multidimensional (74 dimensions) data points. The data compression ratio of this dataset is around 1.63. Note that, not all the 'clustering datasets' in these dataset repositories [24, 25] can be used directly in our experiments as many of these datasets contain non-numeric/missing values for some attributes or the size of the dataset is not large enough to provide any interesting insight.

Synthetic Data

For some real-world, it is possible to achieve even greater compression ratio than the ratio of KDDCupBio04. For instance, the compression ratio of synthetically generated control charts dataset [26] is around 3.80. Unfortunately, the size of this dataset is too small (288 KB) for us to test with. To overcome this limitation, we have generated a synthetic dataset with greater compression reatio consisting of 164 dimensional 145728 data points. These points were distributed evenly among 50 clusters as follows: The 50 cluster centers were sampled from a uniform distribution over the hypercube $[1, 1]^d$. A Gaussian distribution was then generated around each center, where each coordinate was generated independently from a univariate Gaussian with a given standard deviation. The standard deviation varied from 0.01 (very well-separated) up to 0.7 (virtually unclustered). The initial centers were chosen by taking a random sample of data points. The data compression ratio of this dataset is 3.32 using V-PFORDelta coding. Note that the contrasting nature of the chosen real-world and synthetic datasets can provide us an important insight of the effectiveness of our proposed optimization techniques against the dataset of different sizes, dimensions and compression-ratios. The selected datasets can be seen as upper-lower bounds. We can add a few more real-world datasets, but we feel it will only dim the results.



(a) 1 thread

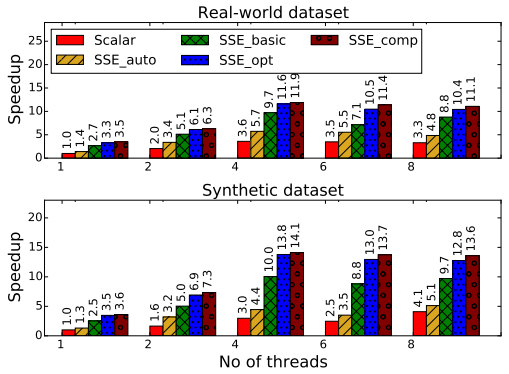
(b) 4 threads

(c) 8 threads

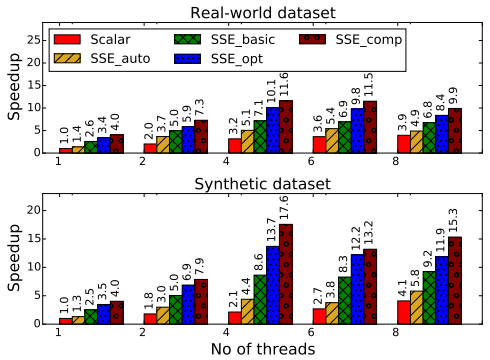
Figure 5: Execution time of the implementations of multi-threaded kernels at different frequencies on HL (Haswell) system.

5.4 Performance Analysis

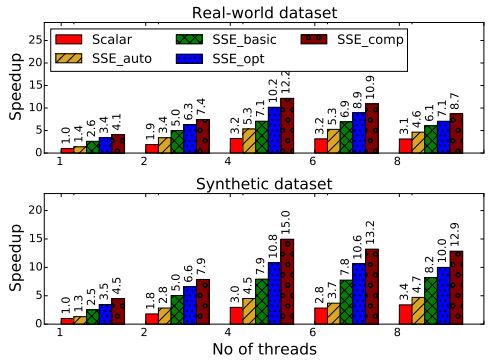
The performance analysis will be carried out in both the Haswell and the KNL systems for the aforementioned *k-means* kernel implementation strategies and features. Figure 5 and Figure 6 present the execution time and speedup of the different strategies when varying the core frequency on Haswell while Figure 7 shows the speedup for KNL. The speedup at a certain core frequency is computed by considering the execution time of the single-threaded kernel as baseline at the same core frequency. To gain further insights into the key drivers of performance variations of different kernel implementations, we also consider certain hardware performance counters provided by PAPI [27]. The list of these counters including the counter-values is presented in Tables 3 and 4.



(a) Speedup on HL (800 MHz)



(b) Speedup on HL (2500 MHz)



(c) Speedup on HL (3500 MHz)

Figure 6: Speedup of the multi-threaded *k-means* kernel implementations at different frequencies on HL (Haswell) system.

We can make several important observations from Figure 5. First, the performance curve in Figure 5a shows that the core frequency has a linear impact on the performance of different kernel implementations as the execution time decreases linearly with the increase of core frequency. For both architectures, running more than one thread per core (SMT) has a negative effects on performance.

It is also shown in Figure 5 and 7 that both *SSE_auto* and *AVX512_auto* kernels provide better performance than the *Scalar* kernel, though the achieved speedup (i.e. 1.4 for SSE and 8.2 for AVX512) is much lower than the ideal speedup (i.e. 4/16). *SSE_basic* kernel clearly outperforms *SSE_auto* kernel for both synthetic and real-world datasets. Having a closer look at the counter values of these two kernel implementations, we can find

Table 3: Average Cache and Memory Related Events for single threaded kernel implementations on Haswell processor

Kmeans Kernel	L1 Data cache accesses		L1 Data cache misses		L1 Data cache write	Stalled cycles on mem. subsystem		Total cycles		Instruction count	
	R	S	R	S	R	R	S	R	S	R	S
Scalar	209967449	288984692659	6307484	8890233722	57397	471240	459710883	363486528	99342163994	1324809657	1834921001955
SSE_auto	51479506	72141873865	6313758	8888548471	56752	2698435	925356343	259319085	73511519337	710537090	990970043470
SSE_basic	55679577	73879158008	6287243	8891312454	37542	1562250	779375650	136502855	39371898023	304937969	406358865890
SSE_optimized	13620296	17981569831	1636397	2240184825	80584	5825528	1528259338	104098777	28227451532	235781815	332783146983
SSE_compressed	13150228	11277606810	1635752	2240555095	70202	195166	283349789	87650731	24933051932	223216928	321490559724

R=*Real dataset* S=*Synthetic dataset*

out that the cache accesses for *SSE_auto* kernel is comparable with that of *SSE_basic* kernel, but the instruction count is doubled for *SSE_auto* kernel over *SSE_basic* kernel. Therefore, we can conclude that compiler auto-vectorization adds some extra instructions in the code that cause a performance penalty when compared with manual vectorization.

Our second observation is that *SSE_optimized* kernel can achieve a speedup of up to 3.6 for single threaded implementation at a peak core frequency (i.e. 3500) on Haswell, which is about 30% better performance over the *SSE_basic* kernel and 6.8 for the KNL. Since in *SSE_optimized* strategy, the total number of memory/cache accesses is further reduced due to in-register *ArgMin* computations and the use of blocked data layout format, the overall performance improvement was expected. This reduction in the number of required cache/memory accesses is apparent in Table 3. The superlinear scaling on the KNL comes from a substancial reduction on the L1D cache misses (Table 4). Since L1D caches from both systems are very similar we can only guess that prefetching is working much better with the new data layout on KNL, but we cannot validate this assumption since we don't have access to that specific performance counter yet. We see the same trend when

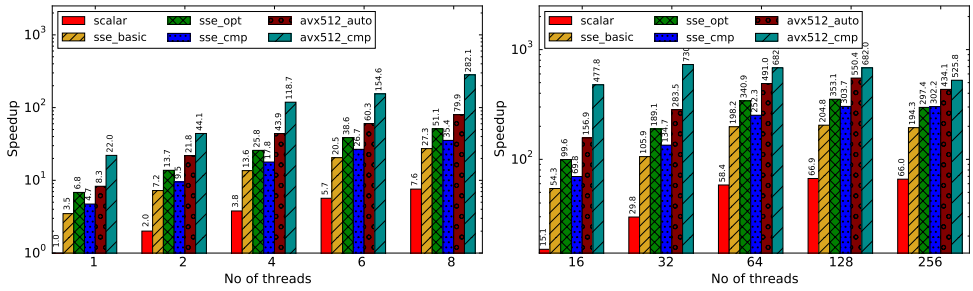


Figure 7: Speedup of the multi-threaded *k-means* kernel implementations on KNL system (log scale).

Table 4: Cache performance counters and instruction count for single threaded kernel on the KNL processor

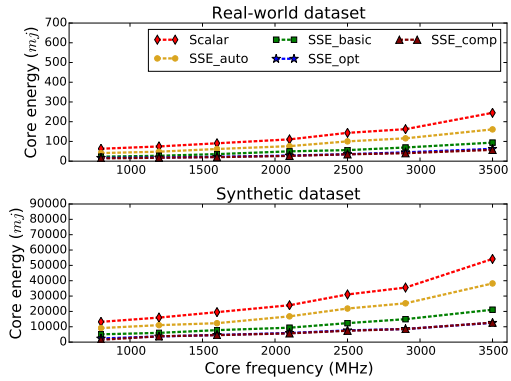
Kmeans kernel	L1 Data cache accesses	L1 Data cache misses	Instruction count
Scalar	292535065167	318225301	1799359093827
SSE_basic	75702075624	439620351	513924495386
SSE_opt	18016121321	84379111	261087333119
SSE_comp	11347608934	85922280	258738990685
AVX512_auto	19327764016	3154904399	129678790549
AVX512_Comp	803823687	61285372	45527089021

comparing *AVX512_auto* and *AVX512_Comp*, with a substantial reduction on both cache accesses and misses.

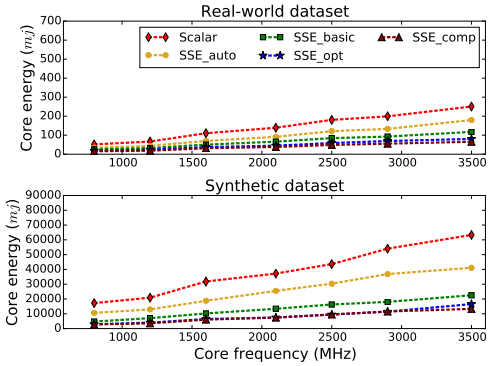
Finally, the *SSE_comp* kernel outperforms all implementations on Haswell, specially when the synthetic dataset is used. As we have already discussed, if the datapoints in the dataset exhibit good correlation among them, the compressed dataset can be used to further reduce the number of memory access. In Table 3, we can observe that the number of memory accesses for synthetic dataset is reduced significantly, which is not the case for the real-world dataset. Therefore, *SSE_comp* does not get much performance benefit for the read-world dataset as the overhead of the decompressing process is not nullified by the reduced number of memory accesses. It is important to note that the performance is increased only at the higher core frequencies, which is reasonable, as the decompression process requires to perform some additional computations. That, and the incredibly low cache miss-rate on the optimized code justifies the "poor" performance of the compressed versions on KNL, since it operates at a low frequency (1.4 Ghz). Still, *AVX512_Comp* achieves a 22*x* speedup over the scalar version. Therefore, we can conclude that the level of speedup for the *SIMD_comp* kernel is sensitive to the compression ratio and core frequency, but has a worst-case performance similar to that of the uncompressed implementation on regular CPUs. It should be worth considering moving the compression to hardware for low frequency architectures.

5.5 Energy Efficiency Analysis

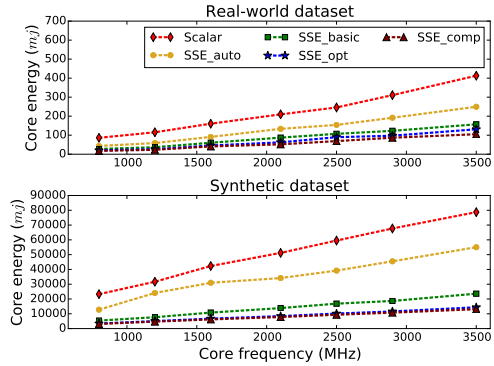
In this section we discuss the implications of the different approaches on the energy efficiency of the analyzed systems. Both Intel Core i7 and Xeon Phi processors have internal counters to estimate the energy consumed by



(a) 1 thread



(b) 4 threads

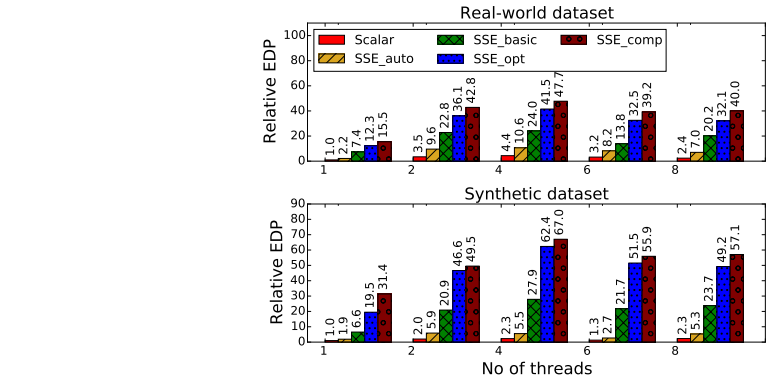


(c) 8 threads

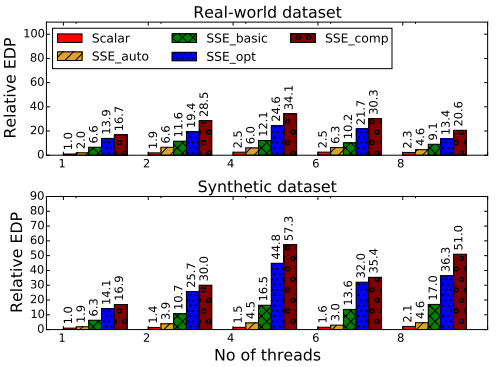
Figure 8: Core energy of the multi-threaded *k-means* implementations at different frequencies on HL (Haswell) system.

different zones of the processor (also known as power planes). We will provide energy measurements for the whole package (including core power and DRAM controller traffic). These counters can be accessed either by the *RAPL* interface (root-level) or the *powercap* interface (user-level).

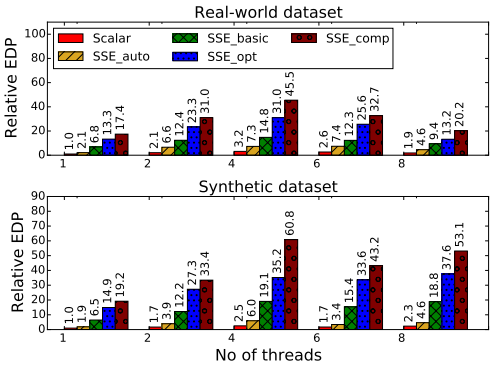
Figure 8 shows the total energy used by the Haswell cores as we vary core frequency and number of threads. Total energy remains similar as we increase the number of cores, meaning that we are not wasting power when adding additional cores in idle time or unprofitable computations. It is also important to note that energy used by the compressed SSE version is very similar to that used by the optimized SSE version. This means that the extra computations performed when compressing/decompressing the data



(a) 800 MHz



(b) 2500 MHz



(c) 3500 MHz

Figure 9: Relative EDP of the multi-threaded *k-means* implementations at different frequencies on HL (Haswell) system.

will burn equal (or less, for high frequencies) energy than the uncompressed version, while performing much better. As for the overall energy reduction, SSE shows improvements in the order of 3.7*x* (14.9*x*) for the real-world dataset and 4.2*x* (16.9*x*) for the synthetic dataset when running on a single (four) thread(s).

When looking at EDP (Figure 9 and 10) we can clearly see the benefits of our proposed implementations. Both the optimized and the compressed SSE (AVX512) versions considerably outperform the scalar codes. SSE-compressed achieves an EDP improvement factor of 10*x* (29*x*) when running the real world (synthetic) dataset on four threads at the lowest frequency we can test. When running at 3.5Ghz, the EDP benefits peak when

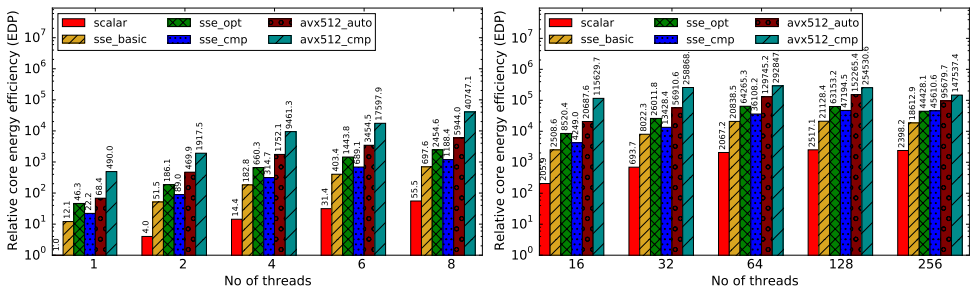


Figure 10: EDP improvements of the multi-threaded *k-means* kernel implementations on KNL system (log scale).

running on four threads with 14*x* and 24*x* EDP improvements over the scalar version running on four threads for the real-world and synthetic datasets (respectively). On KNL, the *AVX512_comp* EDP improvements reach 490*x* on a single thread, with a peak of 292848*x* better EDP when running on 64 threads over the scalar version running on a single thread. This super-linear scaling reveals a high power dissipation of the idle cores of the KNL platform as the governor of the KNL system is set to performance, which forces the CPU to use highest possible clock frequency. Since we do not have root access privilege, we could not change the CPU governor or bind the system processes onto a single core so as to prevent the OS scheduler from keeping the cores busy. Nevertheless, this would be the common case for most end users. Finally, it is also worth mentioning that the compressed codes outperform the optimized codes by a factor of 1.47*x* to 1.72*x* for four threads and real-world/synthetic datasets (respectively) when running at high frequencies (Haswell), but perform similarly at low frequencies (worst for KNL). This is consequent with the performance of the compressed codes at high frequencies.

6 Conclusions and Future Work

Grouping a set of elements that have similar properties to each other than to other elements in a different cluster is a problem present in many fields of applications. This technique can be applied to both integer and floating-point application domains. Pixel coordinates on medical imaging, DNA sequence analysis (Guanine Cytosine Adenine Thymine), multivariate data surveys or IDs in social networks are some examples of the integer domain. In this paper, we present a modified integer *k-means* algorithm that achieves both thread-level and data-level parallelism (vectorization).

We use a new SIMD-friendly data layout that improves data locality. In addition, we also perform an in-register implementation of key functions to minimize data transfers from/to the processor register bank. To further reduce the pressure on the memory subsystem, we improve on the optimized SIMD version to support compressed datasets. Software compression trades computation cycles (+) with memory bandwidth requirements (-). SIMD can compute more data with less instructions, and, if used wisely, become an opportunity to improve on the application data transfers by compressing/decompressing the data.

We have shown that integration of SIMD-based compression is feasible, as long as we can do it in a reasonable time. Results show improvements on performance and core energy consumption of a state-of-the-art *k-means* implementation when running on a single thread by 4.5*x* and 8.7*x* respectively. EDP improvements range between 15*x* to 57*x*, depending on the input set, for an i7 Haswell CPU. On the Xeon Phi KNL architecture results are even better, with ~22*x* improvements on both performance and energy and EDP improvements of 490x for a single thread. Compression will become of critical importance as the use of wide vectors turns CPU bound applications into memory bound, leaving more idle time to compress-decompress (note: Intel 512-bits, ARM-SVE 2048-bits). However, there may be cases where better compression algorithms or hardware support becomes necessary, specially on systems that run below 2GHz, and we are working to solve that issue.

Improving the performance of clustering algorithms improves time to solution, that can be critical in market research and other close to real-time scenarios. In addition, it allows to compute bigger datasets in a "reasonable" time. For example, image processing of medical images for personalized medicine can highly benefit from this, increasing resolution of the images or resonances while producing the output in a similar time frame. On the other hand, improving the energy efficiency translates into a reduction on operation and running costs, a reduction on cooling needs and that usually translates into a reduction on the size of the machinery that computes the algorithms. This can mean a huge improvement on personalized medicine, making PET scans, antibiotic resistance or blood tests more accessible to small clinics.

In future, we would like to extend our study on evaluating the effect of using a look-up table for the approximate computation (precomputed $\langle \vec{c}_i, \vec{c}_i \rangle$ values) in the *labeling state*. Our initial study shows that, the use of precomputed values using look-up table can lead to more than 30% performance improvement for the single threaded *SSE_opt* kernel implementation. It is

also a part of our future work to use compression in other algorithms. In fact we are currently working with compression on B-Trees, in addition to previous work on industrial time series compute kernels. Furthermore, widening the coverage to floating point is a necessary future step. Changing the compression algorithm for one with floating-point support is straight-forward. The feasibility on that domain will depend on the computational requirements of the compression algorithm and the compression ratios achieved.

References

[1] David Arthur and Sergei Vassilvitskii. 'K-means++: The Advantages of Careful Seeding'. In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*. 2007, pp. 1027–1035. ISBN: 978-0-898716-24-5.

[2] Mario Zechner and Michael Granitzer. 'K-Means on the Graphics Processor: Design And Experimental Analysis'. In: *International Journal on Advances in Systems and Measurements* 2.3 (2009), pp. 224–235. ISSN: 1942-261x.

[3] Nigel Stephens. *Technology Update: The Scalable Vector Extension (SVE) for the ARMv8-A architecture*. 2016. URL: `https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture`.

[4] Daniel Lemire, Leonid Boytsov and Nathan Kurz. 'SIMD Compression and the Intersection of Sorted Integers'. In: *Software: Practice and Experience* (Apr. 2015).

[5] Sparsh Mittal and Jeffrey Vetter. 'A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems'. In: *IEEE Transactions on Parallel and Distributed Systems* 99.1 (2015), pp. 1–14.

[6] Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig. 'V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series'. In: *Proceedings of the International Conference on High Performance Computing*. Dec. 2015, pp. 416–425.

[7] S. Lloyd. 'Least Squares Quantization in PCM'. In: *IEEE Transaction Information Theory* 28.2 (Sept. 2006), pp. 129–137. ISSN: 0018-9448.

[8] Sherri Burks, Greg Harrell and Jin Wang. 'On Initial Effects of the K-means Clustering'. In: *Proceedings of the International Conference on Scientific Computing*. Dec. 2015, pp. 200–205.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer and Kevin Skadron. 'A Performance Study of General-purpose Applications on Graphics Processors Using CUDA'. In: *Journal of Parallel and Distributed Computing* 68.10 (Oct. 2008), pp. 1370–1380. ISSN: 0743-7315.

[10] J. Mathew and R. Vijayakumar. 'Enhancement of Parallel K-means Algorithm'. In: *Proceedings of the International Conference on Innovations in Information, Embedded and Communication Systems*. Mar. 2015, pp. 1–6.

[11] Ali Hadian and Saeed Shahrivari. 'High Performance Parallel K-means Clustering for Disk-resident Datasets on Multi-core CPUs'. In: *The Journal of Supercomputing* 69.2 (2014), pp. 845–863.

[12] Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao and Long Gao. 'A Vectorized K-means Algorithm for Intel Many Integrated Core Architecture'. In: *International Symposium on Advanced Parallel Processing Technologies*. Aug. 2013, pp. 277–294. ISBN: 978-3-642-45292-5.

[13] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan and G. Zhang. 'The Design of OpenMP Tasks'. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (Mar. 2009), pp. 404–418.

[14] Hamid Ravaee. 'Finding Protein Complexes via Fuzzy Learning Vector Quantization Algorithm'. In: *Protein-Protein Interactions - Computational and Experimental Tools*. InTech, 2012, pp. 273–284.

[15] Northwestern University, USA. *Parallel K-means Data Clustering*. URL: `http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html`.

[16] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman and Angela Y. Wu. 'An Efficient K-means Clustering Algorithm: Analysis and Implementation'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.7 (July 2002), pp. 881–892. ISSN: 0162-8828.

[17] Greg Hamerly. 'Making K-means Even Faster'. In: *Proceedings of the International Conference on Data Mining*. Apr. 2010, pp. 130–140.

[18] Xiaoli Cui, Pingfei Zhu, Xin Yang, Keqiu Li and Changqing Ji. 'Optimized Big Data K-means Clustering Using MapReduce'. In: *Journal of Supercomputing* 70.3 (Dec. 2014), pp. 1249–1259. ISSN: 0920-8542.

[19] Gang Zeng. 'Fast Approximate K-means via Cluster Closures'. In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR'12. Washington, DC, USA, 2012, pp. 3037–3044. ISBN: 978-1-4673-1226-4.

[20] J. Wang, J. Wang, J. Song, X. S. Xu, H. T. Shen and S. Li. 'Optimized Cartesian K-Means'. In: *IEEE Transactions on Knowledge and Data Engineering* 27.1 (Jan. 2015), pp. 180–192. ISSN: 1041-4347.

[21] Lizhong Xiao, Zhiqing Shao and Gang Liu. 'K-means Algorithm Based on Particle Swarm Optimization Algorithm for Anomaly Intrusion Detection'. In: *Proceedings of the World Congress on Intelligent Control and Automation*. Vol. 2. June 2006, pp. 5854–5858.

[22] R. Mall, V. Jumutc, R. Langone and J. A. K. Suykens. 'Representative Subsets for Big Data Learning Using K-NN Graphs'. In: *IEEE International Conference on Big Data*. Oct. 2014, pp. 37–42.

[23] Raghvendra Mall. 'Sparcity in Large Scale Kernel Models'. PhD thesis. Leuven Arenberg Doctoral School, 2015.

[24] University of Eastern Finland. *Clustering Datasets*. URL: `https://cs.joensuu.fi/sipu/datasets/`.

[25] University of California, Irvine. *Machine Learning Repository*. URL: `https://archive.ics.uci.edu/ml/datasets.html`.

[26] University of California, Irvine. *Synthetic Control Chart Dataset*. URL: `http://archive.ics.uci.edu/ml/machine-learning-databases/synthetic_control-mld/synthetic_control.data.html`.

[27] *Performance Application Programming Interface*. URL: `http://icl.cs.utk.edu/papi/index.html`.