



Norwegian University of  
Science and Technology

# Vectorized Benchmarks for the Berkeley Dwarfs

**Christian De Frene**

Master of Science in Computer Science

Submission date: July 2017

Supervisor: Magnus Jahre, IDI

Co-supervisor: Juan Manuel Cebrian Gonzalez, Barcelona Supercomputing Center,  
Spain

Norwegian University of Science and Technology  
Department of Computer Science



# Problem description

## Vectorized Benchmarks for the Berkeley Dwarfs

Benchmarking is the standard method for conducting scientific experiments in computer science. Researchers decide on a selection of benchmarks which are a representative of applications of interest. These benchmarks are studied in detail to uncover generalized insights that can then be applied to real computer systems. It is critical that the selected benchmarks cover a wide range of software applications to ensure that the uncovered insights have high validity.

Cebrian et al. (ISPASS 2014) observed that vectorized benchmarks change key architectural trade-offs compared to scalar implementations. For this reason, they proposed ParVec which consists of vectorized versions of a subset of the well-known PARSEC benchmarks. Unfortunately, ParVec and PARSEC only partly cover the classes of widely deployed parallel programs (as exemplified by the Berkeley Dwarfs). This is the main motivation for the SIMDwarfs effort which aims to provide the research community with a set of SIMD-enabled applications and kernels that cover the Berkeley Dwarfs.

The student will contribute to the SIMDwarfs effort by adding vectorized benchmark implementations for selected Berkeley Dwarfs. The missing dwarfs are “N-Body Methods”, “Dynamic Programming”, “Backtrack and Branch-and-Bound”, “Graphical Models” and “Finite State Machines”. For each added benchmark, the student should evaluate its performance and scalability as well as identify potential architectural bottlenecks. When possible, the student should improve the benchmark implementation to reduce the impact of the identified bottleneck.

**Advisor:** Magnus Jahre, IDI

**Co-advisor:** Juan Manuel Cebrián González, Barcelona Supercomputing Center (BSC)

# Abstract

In order to guide development of new hardware that meet ever increasing needs, researchers and system designers need high quality performance evaluation tools. In computer science, benchmarking has emerged as one of the most important methods for this purpose. Multiple benchmarks that collectively evaluate a system for a wide range of characteristics in a specific area of interest are compiled into benchmark suites. The purpose is to increase the chances that insight of high validity can be leveraged, i.e. the insight is of high enough accuracy to be applied to real computer systems. The Berkeley dwarfs taxonomy, which are 13 computational patterns in widespread use in the fields of science and engineering, can be used for this purpose.

From the start of the 21st century, as conventional instruction-level parallelism has failed to provide further microprocessor performance increases, the industry has been looking for ways to exploit other types of parallelism as well. One of these is data-level parallelism, found in the single input multiple data (SIMD) computer organization. Research shows that using vectorization can offer more benefits, e.g. increased energy efficiency. However, while SIMD is seeing increased adoption today, [Cebrian, Jahre, et al. 2014] noticed that SIMD-aware benchmarking tools are not as widely available, which they argue can cause SIMD designers to under/over estimate the impact of their contributions. For this reason, they proposed SIMDwarfs, aiming to offer the research community with a SIMD-aware benchmark suite covering all 13 Berkeley dwarfs.

In this thesis we have contributed to SIMDwarfs by analyzing four retrieved, vectorized benchmark implementations from three uncovered dwarfs: `nbody` from n-body methods, `nqueens` from backtrack and branch-and-bound, and `NW` and `SWat` from dynamic programming. All implementations were evaluated using non-vectorized configurations and configurations utilizing SSE and AVX SIMD extensions. The results indicated that while vectorization offered improved performance, the hardware used for the evaluations limited further performance increases. With these implementations added, SIMDwarfs now cover 10 of 13 dwarfs.

# Sammendrag

For å veilede utviklingen av ny maskinvare som møter stadig økende behov, trenger forskere og systemdesignere høyverdige ytelseevalueringverktøy. I datavitenskapen har benchmarking trådd frem som en av de viktigste metodene til dette formålet. Flere benchmarks som sammenlagt evaluerer et system for et bredt spekter av egenskaper i et bestemt område av interesse, blir samlet inn i benchmark-suiter. Hensikten er å øke sjansene for at innsikt av høy validitet kan utnyttes, dvs. innsikt er av høy nok nøyaktighet til å kunne brukes på ekte datasystemer. Berkeleys dvergetaksonomi, som er 13 beregningsmønstre i utbredt bruk innen vitenskap og industri, kan brukes til dette formålet.

Fra begynnelsen av det 21. århundre, da konvensjonell instruksjonsnivå parallelisme har sluttet å gi ytterligere ytelseøkninger for mikroprosessorer, har industrien vært på utkikk etter måter å utnytte andre typer parallelisme. En av disse er data-nivå parallelisme, funnet i single input multiple data (SIMD) datamaskinorganisasjonen. Forskning viser at bruk av vektorisering kan tilby flere fordeler, f.eks. økt energieffektivitet. Imidlertid, mens SIMD ser økt utbredelse i dag, merket [Cebrian, Jahre, et al. 2014] at SIMD-bevisste benchmarkingverktøy ikke er allment tilgjengelige. Dette hevder de kan føre til at SIMD-designere under- eller overestimerer virkningen av deres bidrag. Av denne grunn foreslo de SIMDwarfs, med sikte på å tilby forskningsmiljøet en SIMD-bevisst benchmark-suite som dekker alle 13 Berkeley-dvergene.

I denne oppgaven har vi bidratt til SIMDwarfs ved å analysere fire funnede, vektoriserte benchmarkimplementeringer fra tre udekkede dverger: **nbody** fra n-body methods, **nqueens** fra backtrack and branch-and-bound, og **NW** og **SWat** fra dynamic programming. Alle implementeringer ble evaluert ved hjelp av ikke-vektoriserte konfigurasjoner og konfigurasjoner ved bruk av SSE- og AVX SIMD-utvidelser. Resultatene viste at mens vektorisering tilbød forbedret ytelse, hindrer maskinvaren som brukes til evalueringene, ytterligere ytelsesforbedringer. Med disse implementasjonene lagt til, dekker SIMDwarfs nå 10 av 13 dverger.

# Preface

This is the outcome of my master thesis in Computer Science at NTNU. It is a continuation of my specialization project that I worked on during fall 2016.

# Acknowledgements

I would like to thank my two advisors, Magnus Jahre and Juan Manuel Cebrián González, for all their valuable insight and helpful suggestions on the theoretical and technical aspects of both this thesis and the specialization project that led up to it.

# Table of contents

<b>Problem Description</b>	<b>i</b>
<b>Abstract (English)</b>	<b>ii</b>
<b>Abstract (Norwegian)</b>	<b>iii</b>
<b>Preface</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Table of Contents</b>	<b>vi</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Listings</b>	<b>x</b>
<b>Abbreviations</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Dwarf Coverage Table . . . . .	3
1.3 Research Tasks . . . . .	6
1.4 Contribution . . . . .	6
1.5 Thesis Outline . . . . .	6
<b>2 Background</b>	<b>8</b>
2.1 SIMD and Vectorization . . . . .	8
2.1.1 SIMD advantages and disadvantages . . . . .	8
2.1.2 Applications of SIMD . . . . .	9
2.1.3 Auto-vectorization . . . . .	13
2.1.4 SIMD in heterogeneous systems . . . . .	13
2.2 Benchmarking . . . . .	13
2.2.1 Benchmark Requirements . . . . .	14
2.2.2 Suitable Evaluation Characteristics . . . . .	14
2.3 Other Benchmark Suites . . . . .	16
2.3.1 Suites used by SIMDwarfs . . . . .	16
2.3.2 Miscellaneous suites . . . . .	17
<b>3 Methodology</b>	<b>19</b>
3.1 Profiling . . . . .	19
3.2 Wrapper Library . . . . .	20
3.3 ParVec Framework . . . . .	23
3.4 Benchmark Evaluation . . . . .	24

3.4.1	Input parameters . . . . .	24
3.4.2	Experimental setup . . . . .	24
3.4.3	Evaluation data . . . . .	24
<b>4</b>	<b>Benchmark Analysis: N-body</b>	<b>27</b>
4.1	Algorithm and Vectorization . . . . .	27
4.2	Results and Discussion . . . . .	28
4.2.1	Runtime, Cycle and Instruction Count . . . . .	29
4.2.2	Stalls . . . . .	30
4.2.3	Cache Performance . . . . .	31
4.3	Potential for Further Performance Improvement . . . . .	32
<b>5</b>	<b>Benchmark Analysis: N-queens</b>	<b>34</b>
5.1	Algorithm and Vectorization . . . . .	35
5.2	Results and Discussion . . . . .	36
5.2.1	Runtime, Cycle and Instruction Count . . . . .	36
5.2.2	Stalls . . . . .	38
5.2.3	Cache Performance . . . . .	39
5.3	Potential for Further Performance Improvement . . . . .	40
<b>6</b>	<b>Benchmark Analysis: NW and SWat</b>	<b>41</b>
6.1	Algorithm and Vectorization . . . . .	41
6.2	NW: Results and Discussion . . . . .	42
6.2.1	Runtime, Cycle and Instruction Count . . . . .	42
6.2.2	Stalls . . . . .	44
6.2.3	Cache Performance . . . . .	45
6.3	SWat: Results and Discussion . . . . .	45
6.3.1	Runtime, Cycle and Instruction Count . . . . .	46
6.3.2	Stalls . . . . .	47
6.3.3	Cache Performance . . . . .	48
6.4	Potential for Further Performance Improvement . . . . .	49
<b>7</b>	<b>Conclusion and Future Work</b>	<b>50</b>
7.1	Future Work: Towards Complete Coverage . . . . .	50
7.1.1	Combinational Logic . . . . .	50
7.1.2	Graphical Models . . . . .	51
7.1.3	Finite State Machines . . . . .	51
7.2	The Future of SIMD Extensions . . . . .	51
	<b>Bibliography</b>	<b>53</b>
	<b>Appendices</b>	<b>58</b>
<b>A</b>	<b>Figures for Small and Large Inputs</b>	<b>60</b>
1.1	N-body . . . . .	60
1.2	N-Queens . . . . .	64
1.3	Needleman-Wunsch . . . . .	68
1.4	Smith-Waterman . . . . .	72



# List of Figures

1.1	A comparison of the SIMD and MIMD organizations . . . . .	1
1.2	Relative Performance Per Cycle . . . . .	2
1.3	The Five 'Categories of Vectorization' . . . . .	4
1.4	Chapter Dependencies In This Thesis . . . . .	7
2.1	Scalar vs. Vectorized Operations . . . . .	9
2.2	Vector Operation Using One Lane and Four Lanes . . . . .	11
2.3	Comparison of a typical CPU and GPU organization . . . . .	12
2.4	Arithmetic Intensity . . . . .	16
3.1	Verification process of SIMDwarfs benchmarks . . . . .	19
4.1	2D N-Body Simulation . . . . .	27
4.2	N-Body Running Times for Three Input Sizes . . . . .	29
4.3	N-Body Execution Cycles . . . . .	29
4.4	N-Body Top-Down Cycle Breakdown . . . . .	29
4.5	N-Body Instructions Per Stage . . . . .	29
4.6	N-Body Normalized Stalled Cycles . . . . .	31
4.7	N-Body Normalized Stalls Per Stage . . . . .	31
4.8	N-Body Dispatch Stalls Breakdown . . . . .	31
4.9	N-Body Execution Activity Breakdown . . . . .	31
4.10	N-Body L1 Accesses . . . . .	32
4.11	N-Body L1 Miss Rate . . . . .	32
4.12	N-Body L2 and L3 Accesses . . . . .	32
4.13	N-Body L2 and L3 Miss Rates . . . . .	32
5.1	8-Queens Solution . . . . .	34
5.2	Computation Tree After Partial Breadth-First Search . . . . .	36
5.3	N-Queens Running Times for Three Input Sizes . . . . .	37
5.4	N-Queens Execution Cycles . . . . .	37
5.5	N-Queens Top-Down Cycle Breakdown . . . . .	37
5.6	N-Queens Instructions Per Stage . . . . .	37
5.7	N-Queens Normalized Stalled Cycles . . . . .	38
5.8	N-Queens Normalized Stalls Per Stage . . . . .	38
5.9	N-Queens Dispatch Stalls Breakdown . . . . .	38
5.10	N-Queens Execution Activity Breakdown . . . . .	38
5.11	N-Queens L1 Accesses . . . . .	39
5.12	N-Queens L1 Miss Rate . . . . .	39
5.13	N-Queens L2 and L3 Accesses . . . . .	39
5.14	N-Queens L2 and L3 Miss Rates . . . . .	39
6.1	Sequence Alignment . . . . .	41
6.2	NW Running Times for Three Inputs . . . . .	43

6.3	NW Execution Cycles . . . . .	43
6.4	NW Top-Down Cycle Breakdown . . . . .	43
6.5	NW Instructions Per Stage . . . . .	43
6.6	NW Normalized Stalled Cycles . . . . .	44
6.7	NW Normalized Stalls Per Stage . . . . .	44
6.8	NW Dispatch Stalls Breakdown . . . . .	44
6.9	NW Execution Activity Breakdown . . . . .	44
6.10	NW L1 Accesses . . . . .	45
6.11	NW L1 Miss Rate . . . . .	45
6.12	NW L2 and L3 Accesses . . . . .	45
6.13	NW L2 and L3 Miss Rates . . . . .	45
6.14	SWat Running Times for Three Inputs . . . . .	46
6.15	SWat Execution Cycles . . . . .	46
6.16	SWat Top-Down Cycle Breakdown . . . . .	46
6.17	SWat Instructions Per Stage . . . . .	46
6.18	SWat Normalized Stalled Cycles . . . . .	47
6.19	SWat Normalized Stalls Per Stage . . . . .	47
6.20	SWat Dispatch Stalls Breakdown . . . . .	47
6.21	SWat Execution Activity Breakdown . . . . .	47
6.22	SWat L1 Accesses . . . . .	48
6.23	SWat L1 Miss Rate . . . . .	48
6.24	SWat L2 and L3 Accesses . . . . .	48
6.25	SWat L2 and L3 Miss Rates . . . . .	48

# List of Tables

1.1	Berkeley Dwarfs Table . . . . .	5
2.1	Other Benchmark Suites' Dwarf Coverage . . . . .	18
3.1	Input Parameters For the Evaluated Benchmarks . . . . .	24
4.1	Runtime for Original and Improved Nbody Application . . . . .	28
5.1	N-Queens Solutions up to N=22 . . . . .	34

# List of Listings

1	Code Using Intel's AVX Intrinsics . . . . .	21
2	ParVec Wrapper Library Contents . . . . .	22
3	Code Using the ParVec Wrapper Library . . . . .	22
4	Using the Parsecmgmt Script . . . . .	23

# Abbreviations

- API** Application Programming Interface. A set of methods enabling communication between software components.
- AVX** Advanced Vector eXtensions. 256-bit vector extensions for the x86 architecture.
- CPU** Central Processing Unit. The main processor in a traditional microprocessor.
- DLP** Data-Level Parallelism. Parallelism achieved by performing the same operation on independent data.
- GCC** GNU Compiler Collection.
- GPGPU** General-Purpose Graphics Processing Unit. Running conventional microprocessor applications on a GPU.
- GPU** Graphics Processing Unit. An external or integrated processor specialized for graphics operations.
- HPC** High-Performance Computing. Performing advanced computations efficiently, reliably and quickly.
- ICC** Intel C++ Compiler.
- ILP** Instruction-Level Parallelism. Parallelism achieved by performing multiple instructions independently.
- ISA** Instruction Set Architecture. The set of instructions for a certain computer architecture.
- KNL** Knights Landing. Code name for Intel's second generation Xeon Phi products.
- MIMD** Multiple Input Multiple Data. Computer organization used in a typical multi-processor.
- MMX** MultiMedia eXtensions. 64-bit vector extensions for the x86 architecture.
- MIC** Many Integrated Core. Architecture from Intel aimed at supercomputing.
- MVL** Maximum Vector Length.
- NW** Needleman-Wunsch. Algorithm for global sequence alignment.
- ROB** Re-Order buffer. Helps perform out-of-order execution.
- RS** Reservation Station. Enables fast access to newly computed data values.
- SB** Store buffer. Allows the processor to speculate on store operations.
- SCL** Scalar.
- SIMD** Single Input Multiple Data. Computer organization used for GPUs and vector processors.
- SSE** Streaming SIMD Extensions. 128-bit vector extensions for the x86 architecture.
- SVE** Scalable Vector Extension. Novel SIMD extensions technology from ARM for their architectures.
- SWAR** SIMD Within A Register. Packing processor words with multiple data values that can be operated on in parallel.
- SWat** Smith-Waterman. Algorithm for local sequence alignment.
- UOP** Micro-Operation. Detailed, low-level instruction derived from the processors input, the macro-operation.



# Chapter 1: Introduction

## 1.1 Motivation

Flynn's taxonomy [Flynn 1972] defines four types of computer organizations based on their utilization of streams, i.e. instruction or data sequences that are operated on by the processor. The four organizations are *single input single data* (SISD), *single input multiple data* (SIMD), *multiple input, single data* (MISD) and *multiple input multiple data* (MIMD). Time has rendered both the SISD (uniprocessor) and MISD organizations obsolete, in favor of the more efficient, parallel organizations of SIMD and MIMD. An illustration of the workings of these two is shown in Figure 1.1. SIMD exploits data-level parallelism (DLP), defined by [David A. Patterson et al. 2014, sec. 6.3] as parallelism achieved by performing the same operation on independent data. MIMD has the ability to perform multiple, independent operations on separate data.

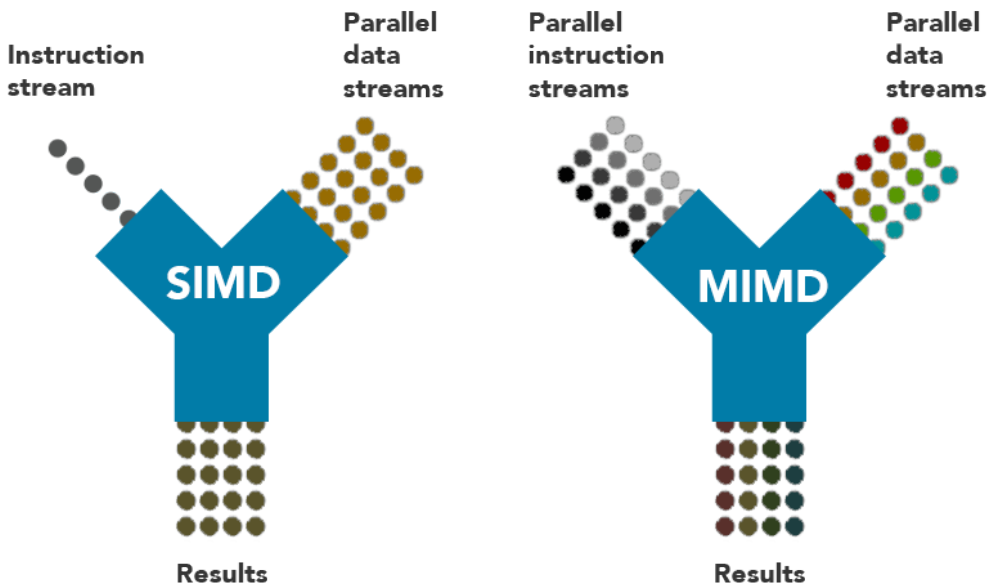


FIGURE 1.1: A COMPARISON OF THE SIMD AND MIMD ORGANIZATIONS

From the 1970's until the early 1990's, SIMD was widely deployed in supercomputers [Stringer 2016]. Various factors, including the rapid performance increase of commodity microprocessors in the 1980's, coupled with the advent of massive parallelism and distributed memory, high energy demands of large supercomputers, as well as lower prices, contributed to a shift in the industry from SIMD, using a few, powerful processors, to MIMD, using a higher number of less powerful processors that overall achieved higher performance [National Research Council and others 2005]. This move is still visible to-

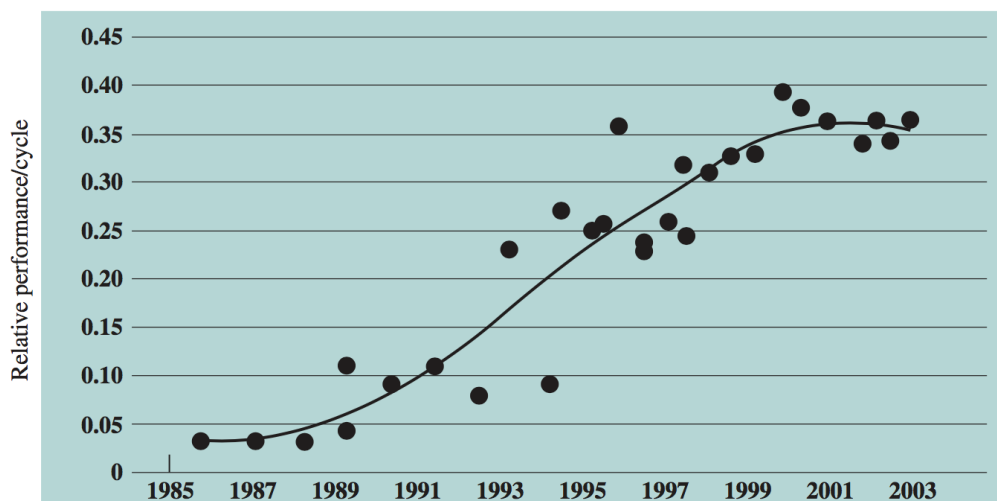


FIGURE 1.2: COMBINED CPU BENCHMARK RESULTS AND CLOCK FREQUENCIES OF INTEL HARDWARE. SOURCE: [STALLINGS 2013, SEC. 18.1]

day, with MIMD being found in most conventional general purpose microprocessors. In addition, [David A. Patterson et al. 2014] suggest that techniques such as instruction-level parallelism (ILP), which has helped deliver steady performance increases, has traditionally disinterested the industry outside of high-performance computing (HPC) to explore other computer organizations.

However, starting from the 2000's, using only ILP has not been able to further increase microprocessor performance. This is shown in Figure 1.2 which graphs the extent to which performance improvement is due to increased exploitation of ILP. Furthermore, Dennard scaling, which state that MOSFET power usage scale downward as we shrink the area, was discredited around 2006 due to current leakage and excess heat buildup severely limiting chip utilization [Esmailzadeh et al. 2011]. This has caused the industry to favor energy efficiency in contrast to performance for microprocessors, and an increasing number of system architects are starting to embrace SIMD because, as [Cebrian, Jahre, et al. 2014] explain, SIMD capabilities are a key component to maximize performance per watt in highly computational intensive applications.

The increased interest in SIMD products and technologies, such as graphics processing units (GPUs), have driven a rapid performance increase that now makes certain SIMD-type processors able to significantly outperform general purpose processors for some tasks [Lee et al. 2010]. This has also led to new developments in existing technologies that enable (limited) SIMD processing on MIMD-type microprocessors, new application programming interfaces (APIs) for GPUs that make them able to perform general purpose computations (GPGPUs), as well as the emergence of heterogeneous systems that try to combine the benefits of SIMD and MIMD, e.g Intel's Xeon Phi products.

One of the most important methods for performance evaluation of computing systems is *benchmarking*. A benchmark is a program chosen to measure performance and resource utilization in specific area of interest, and can provide insight that can be leveraged by scientists to aid the design of new and improved systems [Bienia 2011]. Multiple



benchmarks are compiled into benchmark suites to evaluate various characteristics and behaviors of a system. However, despite the rising popularity of SIMD, fully SIMD-enabled benchmarking tools have been mysteriously absent. [Cebrian, Jahre, et al. 2014] argue that SIMD applications exhibit different behavior than what can be leveraged from existing benchmarking tools, making it hard for the computer architecture community to propose novel techniques. For this reason, they developed the ParVec benchmark suite [Cebrian, Jahre, et al. 2015] by extending a subset of the benchmarks from the PARSEC benchmark suite [PARSEC n.d.] with SIMD capabilities.

Wishing to guide the development of hardware, software, and programming models that efficiently utilize parallelism, a group of computer scientists at Berkeley identified 13 computational patterns, called *dwarfs*, especially important within the fields of science and engineering [Asanovic et al. 2006]. This is a great foundation for a benchmark suite: by covering all dwarfs in this taxonomy, i.e. having at least 13 benchmarks each exhibiting a separate computational pattern, our benchmarking data offers a greater chance to provide us with insight of *high validity*. That is, the data provides a level of accuracy high enough to be applied to the real world.

Since ParVec does not provide complete coverage of the Berkeley dwarfs, Cebrian et al. decided to initiate the SIMDdwarfs project<sup>1</sup> with the following goal: provide the research community with a set of SIMD-enabled applications and kernels that cover all 13 Berkeley Dwarfs. Previously, we have contributed to SIMDdwarfs by performing a literature survey of benchmarks covered by the missing dwarfs. The subsequent sections of this chapter summarize what work was done previously, and how we continued the work over the course of this thesis. More information on SIMD, benchmarking and the Berkeley dwarfs can be found in Chapter 2.

## 1.2 Dwarf Coverage Table

Leading up to this thesis, we carried out a specialization project [De Frène 2016] whose aim was to survey benchmarks covered by dwarfs that we did not have any vectorized benchmark implementations for. The selected benchmarks originated from other, well-known benchmark suites that only partly cover all 13 dwarfs, further discussed in Section 2.3. One of the outcomes of this project was a table that presents each dwarf and what benchmarks are covered by it. Each analyzed benchmark was assigned a 'category of vectorization', ranging from 1 to 5. This number was based on what efforts had been made by others regarding vectorized implementations and literature, thus illustrating the perceived ease or difficulty of adding this benchmark to SIMDdwarfs. The analysis focused only on vectorization, and not other types of parallelism that may be present.

A revised version of the table can be found in Table 1.1. Specifically, two revisions have been made: (1) we have included some new benchmarks, and (2) we have updated the categories of some existing ones. The first revision is based on discovering that new benchmarks had been added to the benchmark suites that inspired SIMDdwarfs. We decided to include them in the table, but have not assigned them a category since we have not surveyed them yet. The second revision is based on upgrading the benchmarks that was analyzed in this thesis, and downgrading some benchmarks that was incorrectly considered implemented. These revisions have been marked in the table with a  $\star$  and a  $\dagger$ , respectively, with a  $\rightarrow$  showing if the change resulted in an adjustment of the highest

---

<sup>1</sup>no documents available yet






-  Category 1: No software has been found
  -  Category 2: Software exists, including original implementation, but without any vectorization
  -  Category 3: Vectorized software exists using manual, automatic or user-directed\* vectorization
  -  Category 4: Vectorized software exists, and uses a wrapper library for vectorization
  -  Category 5: Vectorized implementation added to SIMDwarfs
- \* using tools such as OpenMP and Cilk Plus

FIGURE 1.3: THE FIVE 'CATEGORIES OF VECTORIZATION'

category for one of the dwarfs. We have also rephrased the wording of category 3 to better reflect the multiple ways vectorization can be found in a program. The updated 'categories of vectorization' can be found in Figure 1.3.

TABLE 1.1: THE BERKELEY DWARFS AND THE BENCHMARKS ENVISIONED FOR SIMDWARFS. CATEGORIES HAVE BEEN UPDATED FOR SOME BENCHMARKS BASED ON THE NEW INFORMATION OBTAINED FROM THIS THESIS.

Dwarf	Description	Category	Benchmarks
Dense Linear Algebra	Classic vector and matrix operations where data is stored in array format	5	LU Decomposition, Kmeans, BlackScholes, Dense Matrix Multiplication, Streamcluster, Reduction $\oplus$ , x264 $\oplus$
		3	Vector Operation $\dagger$
		2	Bodytrack $\dagger$ , Strassen $\dagger$ , K-Nearest Neighbors $\dagger$ , Facesim $\dagger\oplus$ , Ferret $\oplus$
			Gaussian Elimination $\star$
Sparse Linear Algebra	Operations on matrices where most elements are zero	5	Fluidanimate $\oplus$
		3	SPMV $\dagger$
		2	Facesim $\dagger\oplus$
Spectral Methods	Data operations in the spectral (frequency) domain	5	Fluidanimate $\oplus$ , FFT
			GPUDWT $\star$
N-Body Methods	Calculations that depend on interactions between (multiple) discrete points	3 $\rightarrow$ 5	N-Body $\dagger$
		2	LavaMD, GEM
Structured Grids	Operations on regular, multidimensional grids	5	Leukocyte, Heart Wall, HotSpot, SRAD1, SRAD2, Particle Filter, Vips, 2D Convolution
		2	Myocyte $\dagger$ , 3D Stencil $\dagger$
			Hotspot3D $\star$
Unstructured Grids	Modelling objects with irregular geometric definitions	5	Histogram, Canneal
		4	Back Propagation
		2	CFD Solver
		1	Unstructured 3D Stencil $\dagger$
MapReduce (previously Monte Carlo)	Repeated, independent executions of a function, aggregating results at the end	5	Swaptions, Raytrace, Histogram, Reduction $\oplus$
			AMCD $\star$
Combinational Logic	Operations exploiting bit-level parallelism to achieve high throughput	5 $\rightarrow$ 3	CRC $\dagger$
Graph Traversal	Traversing objects and examining characteristics	5	BFS, B+Tree, Merge Sort
		3	Quick Sort $\dagger$
		2	Freqmine $\oplus$
		1	Fibonacci $\dagger$
			MUMmerGPU $\star$ , Hybrid Sort $\star$
Dynamic Programming	Computing solutions by solving sub-problems	3 $\rightarrow$ 5	NW $\dagger$ , Swat $\dagger$
		2	Pathfinder
Backtrack and Branch-and-Bound	Dividing the search space into regions and then discarding unsuited ones	3 $\rightarrow$ 5	NQueens $\dagger$
		2	Astar
Graphical Models	Graph data operations, where nodes are variables and edges are conditional probabilities	3	HMM
Finite State Machines	Operations on an interconnected set of states	3	x264 $\oplus$ , Dedup
		2	Freqmine $\oplus$ , Ferret $\oplus$ , TDM
			Huffman $\star$

$\oplus$  Benchmark with different phases, covering multiple dwarfs

$\dagger$  Benchmark that changed category compared to previous table version

$\star$  Benchmark newly added and not surveyed

## 1.3 Research Tasks

The aim of this thesis is to perform analysis of the vectorized benchmark implementations we found during the survey, and then add them to SIMDwarfs. We have formulated the following tasks in order to carry this out:

**Select one retrieved benchmark implementation from the uncovered dwarfs.**

- T1** (Mandatory) Detail how vectorization is applied
- T2** (Mandatory) Determine benchmark performance and scalability
- T3** (Mandatory) Identify potential architectural bottlenecks
- T3.1** (Optional) Detail strategies to reduce the impact of the identified bottleneck
- T3.2** (Optional) Improve the implementation based on the mitigation strategy
- T3.3** (Optional) Determine improved benchmark performance and vector scalability
- T4** (Mandatory) Upload analyzed benchmark to the SIMDwarfs github repository

**Repeat as long as we have time left.**

From the specialization project, we have previously selected the *nbody* benchmark from the N-body methods dwarf, and performed a performance and scalability evaluation. We noticed a possible bottleneck, and laid out a mitigation strategy involving approximating a calculation (more on this in Chapter 4). Following the tasks in this thesis, we have thus completed T2, T3 and T3.1, and are ready to perform T1 and then proceed on to T3.2 and T3.3.

## 1.4 Contribution

The main contribution of this thesis is adding vectorized implementations covering three previously uncovered dwarfs to SIMDwarfs: n-body methods, backtrack and branch-and-bound, and dynamic programming. Four implementations, *nbody*, *nqueens*, *NW* and *SWat* have been analyzed for performance and scalability for three configurations: without vectorization, and vectorized using the SSE and AVX SIMD extensions. *nbody* has also been improved by approximating a high-latency operation. The implementations were ported to the ParVec wrapper library and stored in the SIMDwarfs github repository, which means the code will be reusable for any new ISA that is added to the wrapper in the future.

## 1.5 Thesis Outline

**Chapter 1** introduces the motivation behind this thesis, the need for SIMD-aware benchmarking tools. It also presents work we did on this topic previously, as well as defining research tasks and contributions of this thesis.

**Chapter 2** goes in depth on SIMD and vectorization, benchmarking and what characteristics to evaluate for, as well as presenting other benchmarking suites that inspired SIMDwarfs.

**Chapter 3** presents the process we have followed in order to evaluate the benchmarks, what hardware and software was used, and what data we obtained.

**Chapter 4, 5 and 6** introduces each benchmark that was evaluated, outlines how vectorization is applied, and discusses the results from the evaluation.

**Chapter 7** concludes this thesis, presents what efforts remain in order for SIMDwarfs to achieve full dwarf coverage, and presents some emerging SIMD technologies.

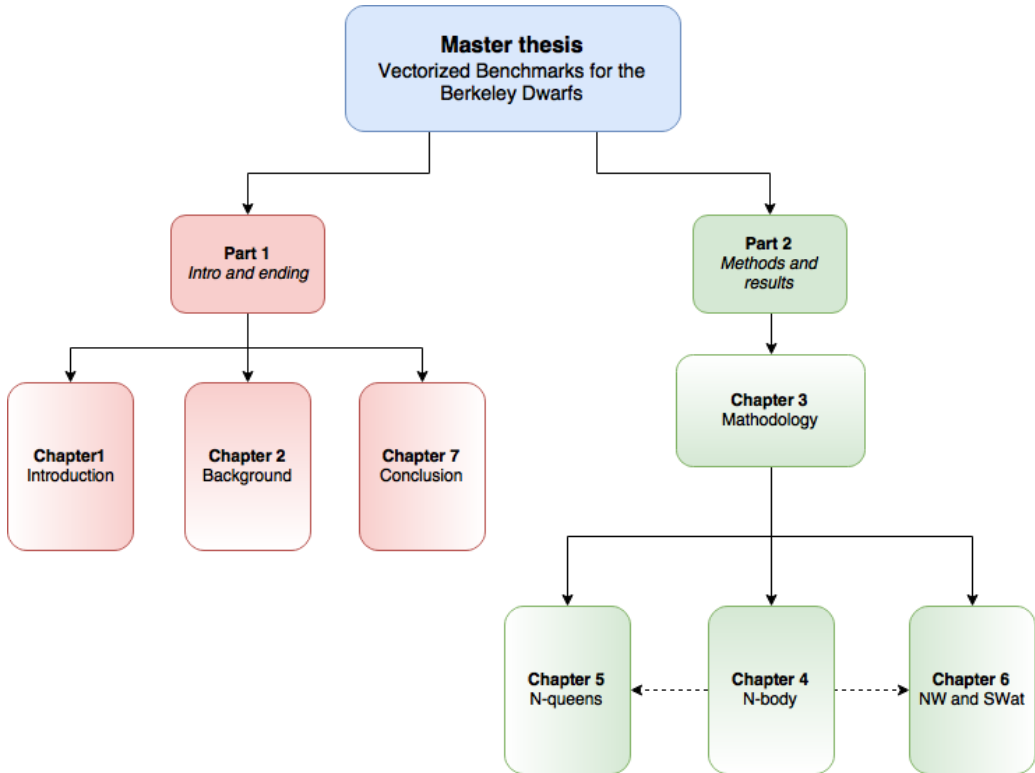


FIGURE 1.4: CHAPTER DEPENDENCIES IN THIS THESIS. CHAPTER 1, 2 AND 7 CAN BE READ WITHOUT MUCH KNOWLEDGE OF OTHER CHAPTERS. CHAPTER 3 PRESENTS THE DATA THAT IS DISCUSSED IN THE FOLLOWING CHAPTERS, AND CHAPTER 4 INCLUDES SOME GRAPH AXIS DETAILS THAT HAVE NOT BEEN REPEATED IN CHAPTER 5 AND 6.

# Chapter 2: Background

This chapter presents the underlying material the work in this thesis is based on. We start by explaining the features of SIMD and vectorization, and their impact. Then we detail requirements for computer benchmarking in order to help system designers in their work. Moving on, we discuss what types of general characteristics are important to evaluate for, such as the Berkeley dwarfs, and then present some SIMD-specific characteristics. Finally, we present some alternative, but lacking, benchmark suites that have been used as an inspiration for SIMDwarfs.

## 2.1 SIMD and Vectorization

Efficient exploitation of DLP relies on the data streams being identically structured. One solution is structuring data inside similarly sized *vectors*, exposing DLP and allowing us to operate on multiple data elements simultaneously. This idea is illustrated in Figure 2.1. In a *scalar* architecture, used in a conventional uniprocessor, one computation outputs one processed data element. While there are multiple ways to increase the throughput rate, this operational pattern stays the same. By exploiting DLP using *vectorization*, a suitable processing element is able to perform the same computation on N individual data elements inside the vector at the same time, also called the *vector length* or *SIMD width*. We will detail various hardware that supports vector processing later on.

[David A. Patterson et al. 2014] corroborate that SIMD works best when the code contains arrays in `for` loops. Using a `for` loop that operates on multiple arrays allows us to unroll the loop and specify operations for each individual array. Thus, given that the arrays contain identically structured data without data dependencies, exploiting DLP can be done effectively. However, there are several factors that can limit SIMD performance. Patterson et al. proceed to state that SIMD is weakest when the code contains case or switch statements where different operations must be performed. They explain that this can cause some execution units to end up containing wrong data, meaning they must be disabled in order for the units with proper data to continue. [Smith et al. 2000] argue that for higher degrees of vectorization, you should vectorize the loops containing these conditional operations, and suggest that performing operations under a mask can help alleviate this issue. Other factors, such as data structure conversions and using slow, horizontal operations (operations on data elements inside the same vector) can also negatively impact the SIMD efficiency.

### 2.1.1 SIMD advantages and disadvantages

Research shows that SIMD offers multiple advantages. From Figure 2.1 we observe that while both computations require one add operation, the vectorized computation has a throughput rate equal to its SIMD width, compared to the scalar computation's throughput rate of 1. This is a substantial advantage of SIMD, as a lower instruction count reduces both cache and pipeline pressure compared to scalar operations [Cebrian, Jahre, et al. 2014].

Moreover, SIMD can offer higher energy efficiency than scalar architectures when

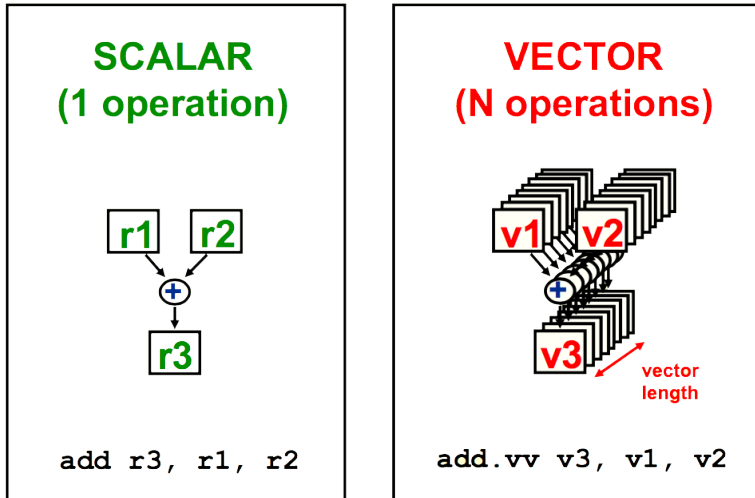


FIGURE 2.1: SCALAR VS. VECTORIZED OPERATIONS. SOURCE: [POZZI ET AL. 2012]

DLP is present. [David A. Patterson et al. 2014, sec. 6.3] mention three characteristics of using vectors that can help save energy: (1) a reduced instruction count means fewer fetch and decode operations, (2) the cost of the latency to main memory is reduced due to fetching whole vectors at a time, and (3) it requires less data hazard checks, since checks are performed for whole vectors and not for each individual data element. [Cebrian, Natvig, et al. 2012] show that applications running on Intel processors dissipate roughly the same average power independently of how many bits were used from the SIMD registers, meaning that there exists an energy saving potential. Furthermore, estimates by [Hennessy et al. 2012, sec. 4.1], assuming an expected biyearly increase of two cores per chip and a doubling of SIMD width every four years, show that towards 2023 the potential speedup from SIMD parallelism is twice that of MIMD parallelism. Based on these results, they argue that SIMD parallelism is at least as important to understand as MIMD parallelism.

Lastly, [Pozzi et al. 2012] state that programming in SIMD allows software developers to achieve parallelism while still thinking sequentially, something that is harder to do when programming in MIMD. This can speed up development times since non-sequential programming requires programmers to adopt a different mindset, which can often be time-consuming.

However, there are some prominent limitations to SIMD. Firstly, it can be arduous to exploit DLP in some algorithms, meaning that SIMD would not be beneficial here. Due to the large amounts of data processed, memory bandwidth needs to be increased, which can lead to greater chip power dissipation when underutilized. There also exists some low-level optimization challenges, e.g. relating to data alignment, that needs to be rectified by hardware manufacturers.

## 2.1.2 Applications of SIMD

Three applications of SIMD have traditionally had the most impact, as covered by [Hennessy et al. 2012]: vector architectures using large vectors that could be operated on in

a non-contiguous fashion, SIMD extensions enabling selected vector operations on short, special vector registers, and GPUs which are mainly used as accelerators for graphics intensive applications.

### Vector architectures

Vector architectures is an early example of SIMD utilization, and quickly became the cornerstone of supercomputing [National Research Council and others 2005]. The first machines supporting a vector architecture started appearing in 1972. This design primarily consists of vector registers, memory that holds the vectors, and fully pipelined vector functional units to process them. Vectors are fixed to a certain size, called the *maximum vector length* (MVL), but a vector register can hold multiple vectors [Pozzi et al. 2012]. Other critical components are registers for scalar values and a combined scalar/vector load/store unit that could operate non-contiguously, meaning that strided accesses were possible. One of the most well-known machines using a vector architecture is the Cray-1 supercomputer from 1975, consisting of eight 4096-bit vector registers and eight 64-bit scalar registers. This computer delivered large amounts of computing power at a price competitive with the most economical computing systems of the day, costing approximately the same as an Apple II [National Research Council and others 2005].

However, the large data processing requirements caused machines with vector architectures to feature large containers to store the memory banks in near vicinity to a few, but powerful, custom made vector processors. This ultimately led to the architecture's downfall. Starting from the 1980's, an influx of cheap and increasingly powerful single-chip microprocessors started to influence the market. This same period also saw improvements to massive parallelism technologies, meaning that it became possible to combine the processing capabilities of thousands of commodity microprocessors that were much easier to scale and both cheaper to buy and operate compared to vector machines. These multi-issue configurations was able to exploit ILP as a substitute for DLP [Koopman 1998], resulting in substantial performance increases. [David A. Patterson et al. 2014] explain that it therefore generally has been few reasons to face the risks of embracing another architecture style such as SIMD.

### SIMD extensions

A more complex approach to SIMD that nonetheless has proved more popular is SIMD within a register (SWAR), found in the SIMD extensions included on many modern processors. Also called subword parallelism, this technique consists of modifying the CPU to perform select, optimized SIMD operations on relatively short, fixed-sized registers. This is done by a vector unit found inside the processor, exploiting the fact that the processor word size is large enough include multiple data elements that can be processed simultaneously. Support for this unit is added as extensions to the instruction set, which decides what operations are possible. The more prominent utilizations today are the MMX, SSE and AVX extensions developed by Intel for the x86 architecture, and the advanced SIMD extensions (NEON) developed by ARM for their products. Originally called multimedia extensions, these extensions started being included in microprocessors in the late 1990's as a way to increase graphics processing capabilities before external or integrated GPUs became commonplace.

SWAR is more intricate, and offer less flexibility than a vector architecture: data



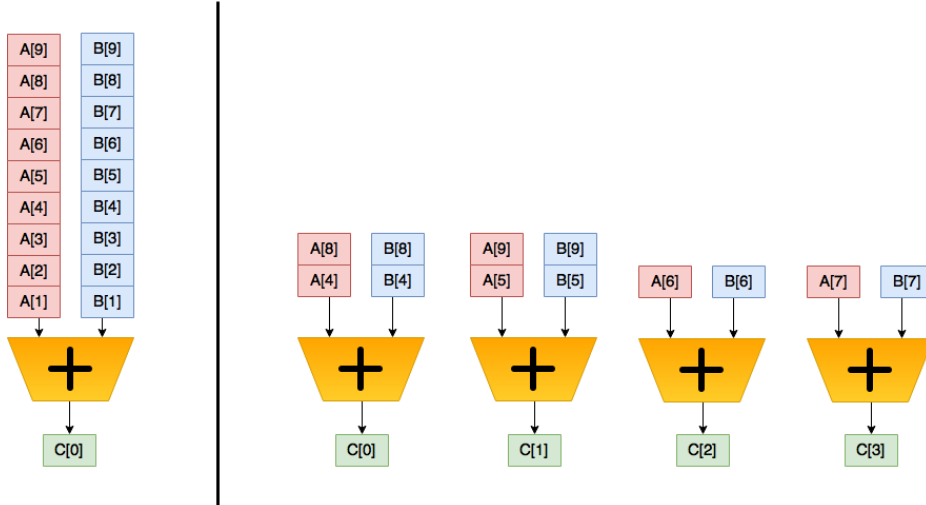


FIGURE 2.2: COMPUTING THE VECTOR OPERATION  $A+B=C$  USING ONE LANE (LEFT) AND FOUR LANES (RIGHT). SOURCE: [DAVID A. PATTERSON ET AL. 2014, SEC. 6.3]

usually need to be loaded as contiguous, whole registers at a time, and there are also fewer operations available. Furthermore, each SWAR opcode need to strictly specify both data type, number of data operands and vector operation, in contrast to vector architectures which had one vector instruction and vector registers that configured automatically based on the data types loaded into it. In the recent years, SWAR developers have tried to incorporate some of these features by adding new operations supporting multiple data type and SIMD width configurations.

Despite the increased complexity compared to operating on a vector architecture, SIMD extensions have become popular in the industry. [Hennessy et al. 2012, sec. 4.3] cite five reasons why, including low cost of implementation and small memory bandwidth requirements in contrast to vector architectures. This interest have caused both register size and quantity to steadily increase each time a new technology has been brought to market: MMX (1997) uses eight 64-bit registers, SSE (1999) utilizes 8 128-bit registers, AVX (2011) can make use of 16 256-bit registers, and the newest AVX-512 (2016) supports up to 32 512-bit registers. Utilizing all registers in the latter two requires a 64-bit system; 32-bit systems support a maximum of 8 registers, regardless of configuration. The latest update to NEON (2015) adds support for 32 128-bit registers, up from 32 64-bit registers previously (2009). [Hennessy et al. 2012, sec. 4.1] state that for computers using the x86 architecture, they expect the register sizes to double every four years. According to [Cebrian, Jahre, et al. 2014], there is a potential for overall speedup directly caused by increasing the SIMD width, known as VL-time performance. Other improvements include the incorporation of multiple pipelined vector functional units, also called *vector lanes*. This concept can be seen in Figure 2.2: by structuring the vectors horizontally, in contrast to vertically, we can compute multiple data elements simultaneously, reducing execution time as well as load/store operations. However, as [Hennessy et al. 2012, sec. 4.1] explain: programmers must be sure to align all the data in memory to the width of the lanes on which the code is run to prevent the compiler from generating scalar

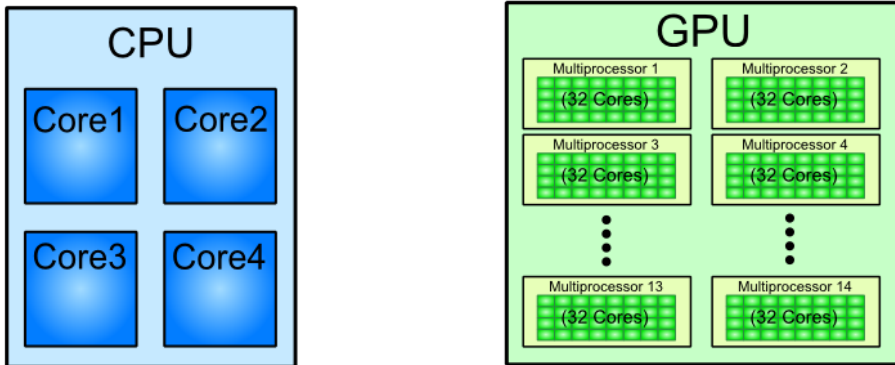


FIGURE 2.3: COMPARISON OF A TYPICAL CPU AND GPU ORGANIZATION. SOURCE: [CPU/GPU ARCHITECTURE COMPARISON N.D.]

instructions for otherwise vectorizable code.

## GPUs

A prominent application of SIMD in the recent years is GPUs. Driven by the big rise in graphical intensive programs such as video games, this product has emerged separate from microprocessor developments and has been motivated by different goals. Today, commodity GPUs are mainly used as accelerators for offloading graphics processing from the CPU. For this reason, most CPUs are designed to be general purpose, while most GPUs are designed to be specialized. GPUs have a focus on throughput rather than latency, manifested in the form of increased computational and memory bandwidth, and favors multithreading rather than employing a multilevel cache hierarchy [David A. Patterson et al. 2014, sec. 6.6]. The result is a processor with better multithreading capabilities across a higher number of processors than what is found in a traditional MIMD CPUs, as shown in Figure 2.3. In a way, GPUs resemble vector processors. Every processor core employ multiple, parallel functional units that process SIMD instructions simultaneously, called SIMD lanes, and structure the data in vectors, though other terms are used, e.g. *warps* on Nvidia products. In addition, they employ multithreading which can complement SIMD: multithreading allows parallelizing work over multiple cores, while SIMD allows parallelizing work within a single core [Microsoft n.d.].

Evaluations show that GPUs significantly outperform CPUs for certain tasks. [Lee et al. 2010] investigated claims that this GPU speedup could reach over 100 times that of the CPU by running 14 kernels on an Nvidia GTX280 GPU and an Intel Core i7 960 CPU. They found that the GPU had a modest, average performance advantage of 2.5x, and suggests that using differing hardware and software, e.g. a mobile CPU with un-optimized code vs. a high-performance GPU with optimized code, can contribute to these abnormally large speedups. Regardless, they argue that software optimizations on the respective platforms are critical to fully utilize compute and bandwidth resources for both CPUs and GPUs, and state that in the absence of such optimizations, CPU implementations are sub-optimal in performance and can be orders of magnitude off their attainable

performance. By using careful multithreading, reorganizing memory access patterns, and applying SIMD optimizations, they found the performance on both CPUs and GPUs is mainly limited by memory bandwidth. Being more bandwidth-focused than CPUs, GPUs have started being employed in the industry for performance purposes.

### 2.1.3 Auto-vectorization

An active research topic in modern compilers is auto-vectorization. As the vectorization capabilities of compilers have advanced, tools that aim to insert vector instructions where it seems practical have emerged. However, while writing manual vector code is a time consuming and error-prone task, auto-vectorization technologies found in modern compilers are not yet able to match the level of vectorization provided. [Maleki et al. 2011] showed this by analyzing the auto-vectorization abilities of the GCC, ICC, and XLC compilers on both synthetic and real applications, and show that 45-71% of the synthetic loops, and only 13-18% of the real application loops were automatically vectorized. It was also found that manual vectorization provided a mean speedup of 2.1x compared to auto-vectorization. In this thesis we have used applications that have been vectorized manually, as this allows us to have better control of which operation are used where, and ensures that SIMD performance is independent of your choice of compiler. Thus, we will not focus further on the aspect of auto-vectorization.

### 2.1.4 SIMD in heterogeneous systems

A system utilizing components that specialize on certain tasks is known as a *heterogeneous* system. Lately, new heterogeneous systems for high performance computing have appeared that try to incorporate SIMD. One example of these products is general purpose GPUs (GP-GPUs), that use special APIs, e.g. OpenCL and CUDA, to program GPUs to perform computations traditionally done by the CPU.

Another novel heterogeneous technology is Intel' Xeon Phi products for high performance computing. At the center of the system architecture, called many integrated core (MIC), is a Xeon Phi co-processor. This enables manycore processing, a new class of multiprocessing that aim to offer higher degrees of parallelism than what was previously possible. The latest generation of Xeon Phi, codenamed Knights Landing (KNL), released in 2016, consists of 72 cores each containing two 512-bit vector units, and is the first product that support AVX-512 SIMD extensions. Intel state that it is capable of handling a wider variety of tasks than traditional accelerators such as GPUs, as well as computing them faster [Intel n.d.(a)].

[Hennessy et al. 2012, sec. 4.1] estimate that combined MIMD and SIMD will yield the highest parallel speedup in 2023, at more than one order of magnitude higher than what individual SIMD and MIMD is able to achieve.

## 2.2 Benchmarking

To design better systems that meet ever increasing needs, system designers and computer scientists use benchmarking, which has become the standard method of performing experiments in computer science [Bienia 2011]. The benchmark results then are studied in detail to uncover generalized insights that can then be applied to real computer systems. A benchmark can take many forms, although [Hennessy et al. 2012, sec. 1.8] argue that

only real-world applications can deliver results of high validity. Other variations, such as kernels (small, key pieces of real applications), toy programs and synthetic benchmarks (applications made to simulate the behavior of a real-world program) are all discredited by researchers, as they make it easy to optimize the hardware and software for these specific operations and thus gain an unfairly high score.

### 2.2.1 Benchmark Requirements

[Weicker 1990] define four requirements of a good benchmark: (1) it is written in a high-level language for portability, (2) it is representative for some kind of programming style, (3) it can be measured easily, and (4) it has wide distribution. Hennessy et al. explain how using compiler flags and source code modifications could be misused, and why they should be considered carefully. They also argue how a guiding principle of reporting performance measurements should be reproducibility, i.e. how other researchers can duplicate the results.

It is critical that the selected benchmarks cover a wide range of software applications, or *workloads*, with various behaviors as to ensure that the uncovered insights have high validity. For this reason, benchmarks of interest are compiled into benchmark suites. [Bienia 2011] define five requirements that should be satisfied in a multithreaded benchmark suite. The first requirement is that the applications should be parallelized, proving that you need to tailor your benchmark suite to an area of interest. The rest are general rules that can be applied to all benchmark suites in general: (2) it should utilize new types of applications that have emerged because of faster hardware, (3) it should contain diverse applications running on a variety of platforms and with different usage models, (4) it should apply state-of-the-art algorithms, and (5) it should be used by researchers.

### 2.2.2 Suitable Evaluation Characteristics

Fully SIMD-aware benchmarking tools have not yet seen wide adoption. By continuing to use benchmarks that provide sub-optimal data and fail to adapt to changes in the industry, system designers can end up doing themselves a disservice. [Cebrian, Jahre, et al. 2014] argue that if benchmarks do not cover the most common architectural features, architects may end up under/over estimating the impact of their contributions. While one issue of designing a SIMD-aware benchmarking tool is that the benchmarks support SIMD, the real challenge lies in determining what features, i.e. characteristics, to evaluate for.

#### System-specific characteristics

[Rabaey et al. 2008] present the problems caused by developing new hardware and software based on extrapolations of existing applications and old data. They argue by doing so, the application community might miscalculate or misinterpret the capabilities of the hardware and software platforms of the future and be lured into dead-ends. A better solution might be formulating new benchmarks that better reflect emerging workloads. Four emergent application areas are presented, as an attempt to fuel development of these 'workloads of the future.' The areas are (1) high-performance computing tasks in scientific fields such as climate research and particle physics, (2) societal IT systems, e.g. relating to automotive or avionic safety and traffic-flow management, (3) personalized, low latency feedback by

societal IT systems and (4) perceptual processing, e.g. voice and virtual reality interfaces that feel natural to the user.

[Dubey 2005] present the Intel Recognition, Mining, Synthesis (RMS) classification, a set of three fundamental processing capabilities Intel deemed necessary for their tera-scale computing platform. The classifications describe how computing systems should *recognize* mathematical models in data sets using machine learning, *mine* the data model in order to extract the relevant data, and then *synthesize* the mined data to draw conclusions that necessarily cannot be found in the model. This classification formed the basis of the PARSEC benchmark suite, which ParVec is a variation of.

A more thorough work on this subject, which was used as the basis of SIMDwarfs, is the Berkeley dwarfs taxonomy. This research stretches back to 2004, when [Colella 2004] defined the *seven dwarfs of high-performance computing*, which were numerical methods important for science and engineering. A dwarf was defined as a an algorithmic method that captures a pattern of computation and communication, specified at a high level of abstraction to so that it can cover a broad range of applications as possible. [Asanovic et al. 2006] based themselves on this taxonomy, and added six more dwarfs by studying three emerging application domains: machine learning, database software, and computer graphics and games. They state that the point of this project is not to identify the low hanging fruit that are highly parallel, but to identify the kernels that are the core computation and communication for important applications in the upcoming decade, independent of the amount of parallelism. The outcome was 13 dwarfs that they hope will guide the development of hardware, software, and programming models that efficiently utilize parallelism. Section 2.3 detail other benchmark suites, in addition to SIMDwarfs, that include benchmarks covering some of the dwarfs.

### Application-specific characteristics

[S. Williams et al. 2009] define the roofline model, a visual tool to compares floating-point and memory performance, and arithmetic/operational intensity (AI). The AI specifies the ratio of floating-point operations per byte of memory accessed, as shown in Figure 2.4. A few example kernels are mapped out on this scale, showing for which of them the intensity scales with problem size (right), and for which the intensity is independent of problem size (left). We recognize many of these kernels as Berkeley dwarfs or as benchmarks considered for SIMDwarfs. This tool can be used to determine which of them are most computationally expensive, which in turn can be used to determine their suitability in a benchmark suite. The roofline model can be used to determine if benchmark performance is *compute-bound*, meaning that processor efficiency will need to be improved in order to increase performance, or *memory-bound*, meaning that the memory subsystem will need to be improved.

[Blem et al. 2011] argue that to sustain the increasing performance developments in GPU processing, architects must continue to address the performance of challenging workloads, instead of using benchmarks that perform well on GPUs. They contribute to this mission by providing and evaluating a list of 'challenge benchmarks' that strain the hardware, and map the key performance limitations. The benchmarks come from existing benchmark suites such as GPGPU-Sim, Rodinia and PARSEC. Benchmarks such as N-queens and Needleman-Wunsch, which have been analyzed in this thesis, as well as benchmarks considered for SIMDwarfs, was deemed sufficiently challenging for GPUs. The authors expect that these results apply to other many-core technologies and vector extensions like Intel's AVX.

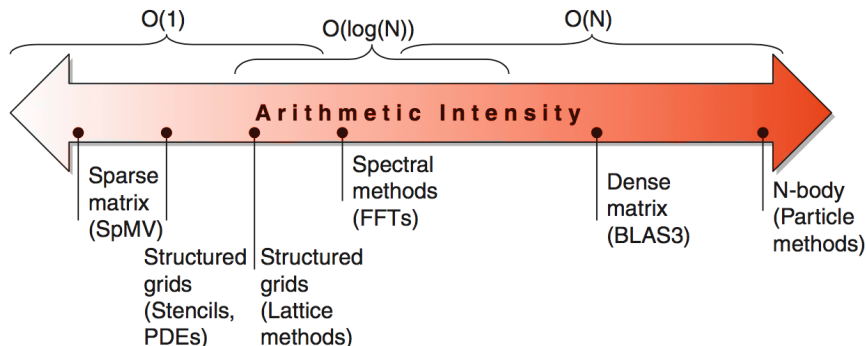


FIGURE 2.4: ROOFLINE MODEL, ARITHMETIC INTENSITY. SOURCE: [HENNESSY ET AL. 2012, SEC. 4.3]

## 2.3 Other Benchmark Suites

In order to select benchmarks for SIMDwarfs that are covered by one or more Berkeley dwarfs, we have drawn inspiration from other, widely adopted benchmark suites. This section focus on four suites, which have supplied the largest amount of potential benchmarks for SIMDwarfs. While all incorporates the dwarf taxonomy to some extent, no one benchmark suite covers all 13 dwarfs. There are also minimal to non-existent SIMD support, meaning the suites all are far from fully SIMD-aware. This section briefly explains their primary purpose and the number of dwarfs covered, with Table 2.1 showing which suites cover which dwarfs. Finally, we mention some miscellaneous suites that have also been considered.

### 2.3.1 Suites used by SIMDwarfs

The Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite [PARSEC n.d.] was released in 2007 in response to the emergence of chip multiprocessors. It was developed as a joint effort between researchers at Princeton and at Intel. The latest version contains 13 parallelized implementations written in C/C++ (10 applications and 3 kernels) covering 8 dwarfs, and has been designed according to the five requirements for benchmark suites defined by [Bienia 2011] in Section 2.2.

[Rodinia n.d.] is intended as a benchmarking tool for heterogeneous systems, aiming to provide insight on accelerators, which no benchmark suite had done before. Released in 2010, it contains benchmarks designed with OpenMP, OpenCL and CUDA. The latest version contains 23 applications, covering 8 dwarfs.

[OpenDwarfs n.d.(a)] provides a benchmark suite consisting of different dwarfs, where the target architectures are multi-/many-core and GPU systems. Benchmarks are realized in OpenCL, and the suite was first made available in 2012. The latest version contains 14 applications covering 12 dwarfs.

The Mont-blanc benchmark suite [Rajovic, Rico, Vipond, et al. 2013] has been made to benchmark the Mont-blanc prototype [Rajovic, Rico, Mantovani, et al. 2016, an HPC system built with commodity hardware. The suite contains 11 applications covering 8

dwarfs, designed with OpenMP and OpenCL.

### 2.3.2 Miscellaneous suites

Additional benchmark suites covering a wide range of disciplines were surveyed. SPLASH-2, SPEC OMP2001 and NAS Parallel Benchmarks (NPB) contain parallel workloads, but with a focus on HPC. SPEC CPU2006 use a wide range of real workloads focusing on processor, memory subsystem and compiler. SPEC CPU2017 is an updated version of CPU2006 including additions such as OpenMP support and energy metrics. Linpack measures floating-point performance, Parboil focus on throughput, and SHOC is used for stress testing and measuring performance of heterogeneous systems. SPEC MPI2007 focus on message passing interface (MPI) parallel applications running on various hardware and software architectures.

We also looked at suites covering specific fields: ALPBench contains parallelized multimedia workloads, and supports vectorization with SSE. BioParallel includes bioinformatic workloads, NU-MineBench measures data mining performance and PhysicsBench is used for computer game physics simulations.

Finally, we looked at the Recursive Benchmark Suite [Ren et al. 2015], which includes recursive, task-parallel benchmarks that have been transformed in order to exploit DLP. Eight applications are included, ranging from microbenchmarks to kernels, that all have been manually vectorized with SSE and AVX-512 intrinsics. This suite includes the n-queens benchmark, which is analyzed in Chapter 5.

TABLE 2.1: OTHER BENCHMARK SUITES' DWARF COVERAGE

Dwarf	Rodinia	PARSEC	OpenDwarfs	Mont-blanc
<b>Dense Linear Algebra</b>	LU Decomposition, Kmeans, Streamcluster, K-Nearest Neighbors, Gaussian Elimination	Streamcluster, BlackScholes, Bodytrack, x264, Facesim, Ferret	Kmeans, LU Decomposition	Dense Matrix Multiplication, Vector Operation, Reduction
<b>Sparse Linear Algebra</b>		Fluidanimate, Facesim	SPMV	SPMV
<b>Spectral Methods</b>	GPUDWT	Fluidanimate	FFT	FFT
<b>N-Body Methods</b>	LavaMD		GEM	N-Body
<b>Structured Grids</b>	Leukocyte, Heart Wall, HotSpot, SRAD1, SRAD2, Particle Filter, Myocyte, Hotspot3D	Vips	SRAD	3D Stencil, 2D Convolution
<b>Unstructured Grids</b>	CFD Solver, Back Propagation	Canneal	CFD Solver	Histogram
<b>MapReduce (previously Monte Carlo)</b>		Swaptions, Raytrace		Histogram, Reduction, AMCD
<b>Combinational Logic</b>			CRC	
<b>Graph Traversal</b>	MUMmerGPU, BFS, B+Tree, Hybrid Sort	Freqmine	BFS	Merge Sort
<b>Dynamic Programming</b>	NW, Pathfinder		NW, Swat	
<b>Backtrack and Branch-and-Bound</b>			NQueens	
<b>Graphical Models</b>			HMM	
<b>Finite State Machines</b>	Huffman	x264, Dedup, Freqmine, Ferret	TDM	



# Chapter 3: Methodology

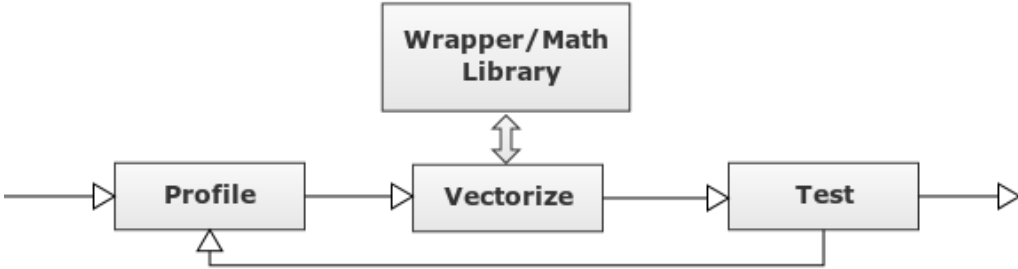


FIGURE 3.1: VERIFICATION PROCESS OF SIMDWARFS BENCHMARKS

Four benchmarks were selected for evaluation and analysis: A particle simulator (**nbody**) covered by the N-body methods dwarf, a combinatorial chess problem (**nqueens**) covered by the Backtrack and Branch-and-bound dwarf, and two sequence alignment algorithms (**NW** and **SW**) covered by the Dynamic programming dwarf.

The verification process for evaluated SIMDwarfs benchmarks is shown in Figure 3.1. Initially, the selected benchmarks are profiled using CodeXL [GPUOpen n.d.]. This is done to determine whether there exist any hotspots in the applications where vectorization can easily be applied. Next, the hotspots are vectorized using the wrapper library and included math libraries. Then, the vectorized benchmark is analyzed and tested to ensure it functions correctly. If not, we iterate by performing the same process until the results are satisfactory. In this thesis, we have performed one iteration for each benchmark. Since we are using applications that have already been manually vectorized, the time spent during the vectorization stage can be greatly reduced.

This chapter details the different stages in the verification process. First, we present the insight from the profiling the applications. Next, the ParVec framework and wrapper library that was used to evaluate SIMD performance across multiple configurations. Then, the input parameter details for each benchmark are described. Finally, we detail specifications of hardware and software that was utilized for the evaluations and provide a summary of the obtained graphs that was used for our data analysis.

## 3.1 Profiling

CodeXL is a collection of tools that can be used to for CPUs and GPUs. It supports time-based sampling, which can expose hotspots in the application where the processor spends large amounts of time, indicating if there are parts in the code that can benefit from vectorization. We performed our profiling using the SSE configuration, as our computer running CodeXL did not support AVX2, which was required for some applications.

For **nbody** we found that one function, *ComputeAccelVec*, accounted for 98% of the hotspot samples. This was expected, as it is here the vectorization is applied. For **nqueens**, two functions, *nqueens\_expand bf* and *nqueens\_block* comprised of 87% and 12%

of the hotspot samples, respectively. The first is the function that generates vectorized work, and the second is the function that processes it. This indicates that the application spends more time creating work than executing it. `NW` and `SWat` are included in the same program. Profiling showed that `searchDatabase` was 85% of the samples for `SW`, while for `SWat`, `opalSearchDatabase` accounted for 81%. Both of these functions are broad with regards to how vectorization is applied, but the results tell us that the two modes utilize different parts of the program.

Based on the hotspot analysis, we translated the benchmarks to the wrapper library and added them as packages in the ParVec framework.

## 3.2 Wrapper Library

To apply the vectorization, all benchmarks are modified to support the ParVec wrapper library. This is a C/C++ library which contains *intrinsics* for SIMD extensions from multiple ISAs. [Intel 2007] define intrinsics as assembly-coded functions that allow you to use C++ function calls and variables in place of assembly instructions. This is useful to C/C++ programmers, who gain direct access to hardware-specific instructions, commonly related to vectorization and parallelization, without having to use assembly. Intrinsics implementing SIMD operations are available for many popular instruction sets, including the x86 SIMD extensions developed by Intel and the NEON extensions from ARM.

The ParVec wrapper library works by defining a common preprocessor macro for equivalent intrinsics across all supported instruction sets, and then translating them to ISA-specific intrinsics at compile-time. Since the translation is performed by the preprocessor, the compiler will still be able to optimize the intrinsics in the code, and thus this mechanism does not add any additional overhead compared to using the intrinsic themselves. [Intel 2007] notes that expanding an intrinsic is done inline, thus eliminating function call overhead altogether. As different compiler can have different auto-vectorization capabilities, applying the vectorization manually allows us more control and more oversight as to what operations are performed. Thus, the purpose of using a wrapper library is to ensure that vectorization is applied consistently across all evaluated benchmarks, independently of what compiler you are using. It also enables us to reuse the same vectorized codebase for multiple target architectures simply by changing the value of the `-c` flag when running the `parsecmgmt` script. Furthermore, should we add support for a new instruction set, the same codebase will work for this configuration as well.

Currently, the library contains intrinsics for Intel's SSE (all versions), AVX, AVX2 and AVX-512 SIMD extensions for both single and double-precision values, as well as the single-precision variant of ARM's NEON SIMD extensions. Other releases, such as ARMv8-A (double-precision NEON) and ARM scalable vector extensions (SVE), are not yet supported. The first is straightforward to add support for, as it consists of additions to the previous intrinsics supporting longer vectors and data types. The latter is more challenging, as no intrinsics are currently available, and there is limited compiler support. Furthermore, the code structure is different, i.e. loop iterations are based on `while` loops, not `for` loops. Thus, we need to refactor the SIMDwarfs codebase in order to support it.

An example of how to use the ParVec wrapper library is shown below. Listing 1 displays code for calculating the following inverse square root formula, implemented using Intel AVX intrinsics.

$$\frac{1}{\sqrt{\text{dist}xVec^2 + \text{dist}yVec^2}}$$

These intrinsics come from the `immintrin.h` file that is bundled with our compiler. Note that `_mm256` at the start of the intrinsic means to output a 256-bit vector, and the `_ps` at end means that the data is to be *packed* inside the vector as *single-precision* (32-bit) floating point values, according to Intel’s terminologies. The `_m256` specifies the data type, saying that the `sqrRecipDistVec` variable holds a 256-bit vector containing single-precision data.

```

1  #include <immintrin.h>
2  __m256 sqrRecipDistVec = _mm256_div_ps(_mm256_set1_ps(1.0),
   ↪  _mm256_sqrt_ps(_mm256_add_ps(_mm256_mul_ps(distxVec,distxVec),
   ↪  _mm256_mul_ps(distyVec,distyVec))));

```

LISTING 1: CODE USING INTEL’S AVX INTRINSICS

We port vectorized applications to the ParVec wrapper library by replacing all intrinsic functions and data types with their library equivalent. An example of how the wrapper library structures the different intrinsics can be viewed in Listing 2. It should be noted that this is an overly simple example, omitting many details for the sake of brevity. The real wrapper library has over 2000 lines of code containing many control structures and workarounds to ensure a high level of portability between the supported architectures and various configurations. More details on the wrapper library are available from [Cebrian, Jahre, et al. 2015].

The final, ported code can be viewed in Listing 3, where the wrapper library is imported through the file `simd.defines.h`. Note that the code also has been transformed from single-precision to precision-agnostic, meaning that we can compile it for both single or double precision, should the wrapper support it.

```

1  #if defined (PARSEC_USE_SSE) || defined (PARSEC_USE_AVX)
2  #include <immintrin.h> // ALL SSE and AVX
3  #endif
4
5  #ifdef PARSEC_USE_SSE
6  #define _MM_TYPE      __m128
7  #define _MM_DIV      _mm_div_ps
8  #define _MM_SET(A)   _mm_set1_ps(A)
9  #define _MM_SQRT     _mm_sqrt_ps
10 #define _MM_ADD      _mm_add_ps
11 #define _MM_MUL      _mm_mul_ps
12 #endif // PARSEC_USE_SSE
13
14 #ifdef PARSEC_USE_AVX
15 #define _MM_TYPE      __m256
16 #define _MM_DIV      _mm256_div_ps
17 #define _MM_SET(A)   _mm256_set1_ps(A)
18 #define _MM_SQRT     _mm256_sqrt_ps
19 #define _MM_ADD      _mm256_add_ps
20 #define _MM_MUL      _mm256_mul_ps
21 #endif // PARSEC_USE_AVX
22
23 #ifdef PARSEC_USE_NEON
24 #include <arm_neon.h> // ALL NEON instructions
25 #define _MM_TYPE      float32x4_t
26 #define _MM_DIV      vdivq_f32
27 #define _MM_SET(A)   vdupq_n_f32(A)
28 #define _MM_SQRT     vsqrtq_f32
29 #define _MM_ADD      vaddq_f32
30 #define _MM_MUL      vmulq_f32
31 #endif // PARSEC_USE_NEON

```

LISTING 2: PARVEC WRAPPER LIBRARY CONTENTS

```

1  #include "simd_defines.h"
2  _MM_TYPE sqrtRecipDistVec = _MM_DIV(_MM_SET(1.0),
  ↪  _MM_SQRT(_MM_ADD(_MM_MUL(distxVec,distxVec),
  ↪  _MM_MUL(distyVec,distyVec))));

```

LISTING 3: CODE USING THE PARVEC WRAPPER LIBRARY

### 3.3 ParVec Framework

The analysis of the benchmark applications has been performed using the ParVec benchmarking framework, which is the same one used for the PARSEC benchmark suite [PARSEC n.d.]. As well as being a powerful overall tool for building, running and evaluating benchmarks, we have access to the useful *hooks* feature, which is used to trigger the measurement tools in the framework. By specifying 'regions of interest' in the code, we can define the critical sections of the program that we want the framework to measure, and omit other, unimportant sections, e.g. those relating to initialization, cleanup and input/output. To enable metrics that give us relevant insight of the SIMD utilization, we use hooks in combination with the performance counter tool PAPI [ICL n.d.], which enable us to record specific CPU events, e.g. execution cycles and cache utilization. For our analysis, we record 54 events across 16 event groups, which are collections of events that are presented together.

```
./parsecgmt -a {build,run} -p parsec.simd.{nbody,nqueens,swat} -c
↪ gcc-{'sse,avx,avx2'}-hooks -i sim{small,medium,large}
```

LISTING 4: USING THE PARSECMGMT SCRIPT

Performing tasks with the framework is done using the `parsecgmt` script. By specifying different flags, or options, we can perform all our required operations. [Bienia 2009] shows the full list of supported options. To obtain the results in this thesis, we have run the script using the options shown in Listing 4. The `-a` chooses the *action* we want to perform, in this case building or running a benchmark executable file. The `-p` selects the appropriate benchmark *package*, which is a directory that contains the source code and support libraries, plus a makefile that explains how the framework should compile and build the executable. It also stores all previously compiled executables so they can be re-used later on. The `-c` specifies which benchmark *configuration* we want, be it vectorized or not. This option tells the framework to load the appropriate support libraries for that specific instruction set, which compiler to use, and whether to enable hooks or not. Finally, the `-i` determines which *input* we want to run the benchmark with. This option itself only supports a select few `.runconf` files, which are located inside each benchmark package. Inside these files we specify the command line arguments to select our desired input parameters for that specific benchmark application. For this thesis we have utilized the three input sizes, further discussed in Section 3.4.

## 3.4 Benchmark Evaluation

### 3.4.1 Input parameters

The benchmarks were evaluated for three input sizes (small, medium and large) as to check if they were input sensitive. Input parameter details are shown in Table 3.1. `nbody` and `nqueens` only take integer values as input. `NW` and `SWat` need a query and database sequence to compare, further explained in Chapter 6. We selected sequences from the 'Swiss-Prot', downloaded from the Universal Protein Resource (UniProt) knowledgebase [UniProt n.d.], which contain manually reviewed sequences of many sizes.

TABLE 3.1: INPUT PARAMETERS FOR THE EVALUATED BENCHMARKS

Benchmark	Small	Medium	Large
nbody	2000 timesteps		
	250 bodies	500 bodies	750 bodies
nqueens	Block size: 15		
	N=11	N=13	N=15
NW	<i>Query:</i> Arbitrary sequence from Swiss-Prot with 325 residues		
	<i>Database:</i> First 1000 sequences from Swiss-Prot	<i>Database:</i> First 5000 sequences from Swiss-Prot	<i>Database:</i> First 10,000 sequences from Swiss-Prot
SWat	<i>Query:</i> Arbitrary sequence from Swiss-Prot with 60 residues		
	<i>Database:</i> First 1000 sequences from Swiss-Prot	<i>Database:</i> First 5000 sequences from Swiss-Prot	<i>Database:</i> First 10,000 sequences from Swiss-Prot

### 3.4.2 Experimental setup

All benchmarks have been analyzed by running scalar, SSE and AVX configurations, built using GCC version 6.2 with the `-O2` flag. The evaluations were performed on a system with an Intel Xeon E5-2640v3 @ 3.4GHz (Haswell) CPU and 130 GB RAM, running Ubuntu 16.04.2 with Linux kernel 4.4.0-78. It supports all versions of SSE as well as AVX and AVX2. PAPI version 5.5 was used to provide performance counters, and the program `cpuset` was used to shield the evaluations from other processes on the machine. We ran each configuration 100 times per input size for each of the 16 event groups.

### 3.4.3 Evaluation data

A script was used to extract the results data and graph it using `pychart`. When generating the graphs, a 0.3 trimmed mean was calculated to remove any outliers. These figures provide us with various insight on how the applications performed on our system, and most are available with both absolute and normalized values. For the data analysis, we have selected these graphs that present us with the most relevant insight:

- **Runtime:** Running time across all configurations and input sizes, which tells us if overall runtime is affected by varying vector and input sizes.

- **Execution Cycles:** Total execution cycles, and the number of cycles that were stalled, for each stage in the instruction pipeline. We can observe if a certain stage raise the number of stalls and if the three configurations function correctly by requiring less cycles than the previous one. Also available in normalized format, showing the ratio of stalls to running cycles for each pipeline stage.
- **Top-down Cycle Breakdown:** A breakdown of what happened in each execution cycle. This is based on the top-down model, presented by [Yasin 2014] as a practical method to quickly identify *true* bottlenecks in out-of-order processors. Based on CPU events, the model segments execution cycles into four basic categories: *retiring*, meaning the cycle finished as expected; *bad*, meaning the cycle stalled due to a failed speculation; *frontend*, meaning a stall occurred during the fetch or decode stages, and *backend*, meaning a stall occurred during the issue, execution or writeback stages. From these categories, an analysis hierarchy is available, whose purpose is to help locate specific performance bottlenecks fast and reliably.
- **Instructions Per Stage:** A count of the total instructions per configuration and the number that retired (finished successfully). We can also view how many micro-operations (UOPS) that were issued, executed and retired. UOPs are detailed, low-level instructions derived from the macro-operations sent to the processor. Since SIMD applications require fewer operations as the vector length increase, we can expect both the macro- and micro-instruction count to decrease accordingly. Some graphs can show the number of retired UOPs to be higher than the issued or executed ones; this is due to a phenomenon called UOP fusion, explained below.
- **Normalized Stalled Cycles:** The ratio of stalled to running execution cycles when there are pending cache accesses to L1, L2, and all levels, respectively. This data can tell us if any stalls were caused by cache misses at that specific level.
- **Dispatch Stall Breakdown:** A breakdown of stall causes during dispatch, a stage in the out-of-order pipeline that deals with specifying what operations should be performed. Based on the CPU events, four causes are given: reservation station (RS), store buffer (SB), reorder buffer (ROB), or stalls elsewhere in the system. A high number of the three first can tell us that the stalls are caused by a lack of hardware resources. Other stalls tell us other parts of the system caused the stalls, e.g. a lack of physical vector registers.
- **Execution Activity Breakdown:** The ratio of stalled to running execution cycles in general, as well the ratio of stalled to running cycles when there were pending L1D misses and pending L2 misses, respectively. This shows the correlation between the total stall rate and stall rate when there were cache misses pending, which can be used to determine if it was the cache misses that caused the stall.
- **L1 Accesses:** The total number of cache accesses reading from L1D, writing to L1D, and reading from L1I. We can view how many accesses were hits, and how many were misses, as to determine cache performance. Also available in normalized format, showing the ratio of hits to misses for each cache. It should be noted that the Intel processor we have used performs *UOP fusion*, an optimization technique present because the instructions need to be translated from a CISC system format to a RISC internal format. The technique works by placing the translated instructions in L1I, but addressing them virtually. While faster and offering more benefits than accessing them from the L1I cache, it poses some challenges to our use of performance counters: the processor accesses the L1I cache once per instruction, which is a compulsory miss to fetch it, and afterwards move the translated UOPs to the virtual UOP cache and access them there, which is not counted. The result is that the few L1I accesses we register will all be misses. More information on UOP

fusion is available from [Fog 2017a].

- **L2 and L3 Accesses:** The total number of cache accesses reading from L2, writing to L2, and reading and writing to and from L3. These graphs also show how many accesses were hits and how many were misses, and are available with normalized values as well.



# Chapter 4: Benchmark Analysis: N-body

The `nbody` benchmark considers the development of a system of  $n$  *bodies* in a 2D space, based on the gravitational forces each body exerts on the other bodies in the system [Koby n.d.]. The forces act according to Newton's universal law of gravitation:

$$F = G \frac{m_1 m_2}{r^2}$$

The law states that an equal force attracts two objects based on a gravitational constant, the combined mass of the objects, and the distance between them. Once we know the forces, we can use Newton's second law to calculate the acceleration of each body in the system, which can be used to determine their velocity and position. N-body simulations, illustrated in Figure 4.1, are used to track the evolution of velocities and positions in such a system, given a number of bodies and time steps. Such a simulation can be applied to bodies of varying size, ranging from celestial bodies in space to atoms in a gas cloud.

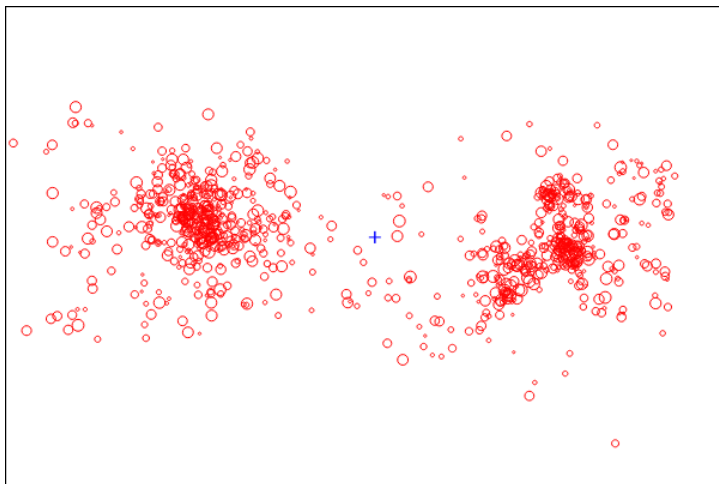


FIGURE 4.1: 2D N-BODY SIMULATION. SOURCE: [N-BODY SIMULATION N.D.]

## 4.1 Algorithm and Vectorization

Without optimizations, an N-body algorithm have to compute the forces of  $N$  bodies for the rest of the bodies in the system, resulting in a quadratic time complexity of  $\mathcal{O}(N^2)$ . Thus, doubling  $N$  quadruples the asymptotic time complexity. However, as the forces between two bodies is the same, the amount of actual computations can be reduced to

$\frac{N^2}{2}$ . Approximations exist that recursively sum the forces of nearby bodies for use in the remaining computations.

Our vectorized `nbody` application, provided by [Davies n.d.], apply no such approximation techniques. The acceleration computation is done by structuring the individual body parameters into arrays and looping through them. This computation has been vectorized manually, using SSE or AVX intrinsics to compute 2 or 4 accelerations simultaneously if using double-precision (double data types), or 4 or 8 if using single-precision (float data types). In our previous project, we analyzed the application and found a bottleneck caused by two intrinsics with high latencies, namely a divide (`div`) and square root (`sqrt`) operation that is used to calculate the inverse square root required for the force computation. Together, they require a combined latency of 26-36 cycles for SSE and 47-64 cycles for AVX, while running on our evaluation system (Haswell). Davies report that these two operations account for 67% of the acceleration computation’s total cycles.

We proposed a mitigation strategy involving approximating this function for this thesis. In a first approach we utilized the inverse square root (`invsqrt`) intrinsic, which is included in Intel’s Short Vector Math Library (SVML). A comparison between the original intrinsic (`1/sqrt`) and `invsqrt` resulted in no significant change in runtime. This suggests that they are both performed in a similar way. An alternative to this approach is to use the reciprocal (`rcp`) intrinsic in combination with the `sqrt` intrinsic. This also gave no significant speedup, and did not include a double-precision variant.

The third approach we tried was using a reciprocal square root approximation, the `rsqrt` intrinsic, which is only available for single-precision. On a haswell system, this operation requires only 5 cycles latency for SSE and 7 cycles latency for AVX. We performed 10 runs with the original and our improved implementation for scalar, SSE and AVX configurations using single-precision, and calculated the mean runtime. The results, viewed in Table 4.1, show that the `rsqrt` approximation gives roughly 25% speedup for SSE and roughly 20% speedup for AVX compared to the original application. This implementation was used for the evaluation.

TABLE 4.1: RUNTIME (SECONDS) FOR ORIGINAL AND IMPROVED NBODY APPLICATION (500 BODIES, 2000 TIMESTEPS. 10 RUNS MEAN, SINGLE-PRECISION)

Implementation	SSE	AVX
Original ( <code>1/sqrt</code> )	0.6107760	0.5773927
Improved ( <code>rsqrt</code> )	0.4544663	0.4576816

## 4.2 Results and Discussion

The data shows that we achieve significant speedup from scalar by using the SSE configuration, but that hardware resources limit the AVX configuration from improving upon this. The data indicates this application is not input sensitive, as shown by the additional graphs in Section 1.1.

The likely culprit of the bottleneck is a high amount of backend stalls (issue, execution and writeback), where the issue stage is the one out of these three where the stalls are most prominent. There are almost 100% RS stalls during dispatch for all configurations, suggesting that the RS included with our hardware is not capable enough to handle

this type of computation efficiently. [Shimpi 2012] show that our system includes 56 RS entries. There are relatively low cache miss rates across all levels, indicating that this application is not memory bound, i.e. we can not improve the algorithm’s performance by improving the memory subsystem.

The data is analyzed and discussed in greater detail below.

### 4.2.1 Runtime, Cycle and Instruction Count

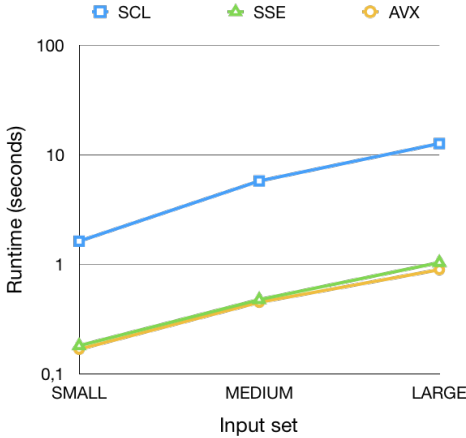


FIGURE 4.2: N-BODY RUNNING TIMES FOR THREE INPUT SIZES (LOG SCALE)

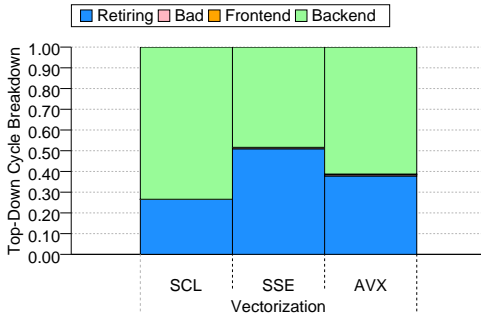


FIGURE 4.4: N-BODY TOP-DOWN CYCLE BREAKDOWN (MEDIUM INPUT)

Figure 4.2 graphs the runtime in seconds for the three configurations over three input sizes. We see from the similar slopes of the three curves that that configurations scale very well between the different inputs. There’s also a substantial speedup from scalar to SSE, but a minuscule speedup between SSE and AVX. There seems to be some stalls between the SSE and AVX configurations.

The low reduction between SSE and AVX goes for execution cycles as well. Figure 4.3 shows the total execution cycles required for each stage in the pipeline, grouped and compared for each of the three configurations. The x-axis presents each pipeline stage in sequential order, with a histogram post for each configuration. The y-axis notes how

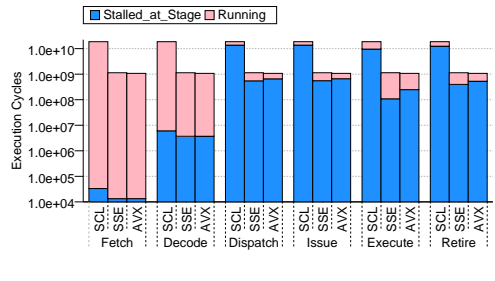


FIGURE 4.3: N-BODY EXECUTION CYCLES (MEDIUM INPUT, LOG SCALE)

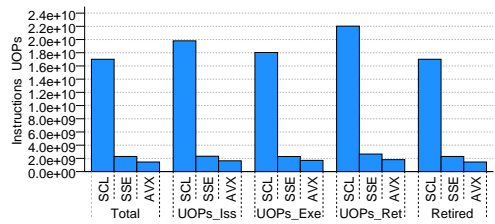


FIGURE 4.5: N-BODY INSTRUCTIONS PER STAGE (MEDIUM INPUT)

many execution cycles were performed by this configuration at this particular stage, and also shows how many of these cycles were stalled. We notice that the number of cycles scale from scalar to SSE, but that the SSE and AVX configurations have a similar cycle count. A cause might be stalls for AVX that is increasing the cycle count, or failure to utilize the increased vector length.

Figure 4.4 shows a cycle breakdown utilizing the top-down model. The x-axis designates each configuration, and the y-axis shows the distribution of the four main categories. We see more retiring instructions for SSE than AVX, which might explain the same runtime and execution cycles. For both the scalar and AVX configurations, backend stalls are dominating. Using the top-down model, we observe that these stalls are caused by either waiting for the memory subsystem to complete, or a combination of high instruction latencies and poor utilization of hardware resources.

The total number of instructions and micro-operations (UOPs) are shown in Figure 4.5. The x-axis shows the three implementations over five stages. The leftmost and rightmost stages show the total number of issued and retired instructions, respectively. The three middle stages show us how many micro-operations the processor issues, executes and retires. The y-axis shows how many instructions were counted. We notice that the SSE configuration reduces instruction count with almost 90% compared to the scalar one. This might indicate that there are some other optimizations done to the SSE configuration that allows it to further reduce its instruction count other than using data-parallel operations. Another possibility can be that it is the scalar configurations that does not scale at the same rate as the vectorized configurations. The AVX configuration requires about half the instructions of the SSE, which seems to scale correctly.

## 4.2.2 Stalls

From Figure 4.6 we can view the share of stalls when there were pending cache accesses. On the x-axis we have all requests that are pending to enter the L1, L2, and all cache levels (L1 through L3), respectively, with bars for each configuration. The y-axis shows the share of cycles that was ultimately stalled. We observe that there are almost no stalls while waiting for L1 and L2 data, which means that any stalls are probably not caused by L1 and L2 cache misses, but must be coming from somewhere else. The high rate of stalls in Any\_Pending is likely due to the the high amount of compulsory misses in L1I caused by UOP fusion.

We will now try to figure out if stalls are constant or if they rise during a particular stage. Figure 4.7 shows the rate of stalled and running cycles for each operational pipeline stage. On the x-axis, we have each pipeline stage with a bar for each configuration. The y-axis shows the percentage of the total running and stalled cycles at that stage. We notice that there is a heavy rate of stalls during the dispatch, issue and retire stages. However, the SSE configuration has consistently fewer stalls than the other two; another way to view it is that the AVX configuration has more stalls than what we should expect from the vector length. The latter seems to be the more probable, given that the top-down showed a higher amount of backend stalls for this configuration compared to the SSE one.

Next, we take a look at the dispatch stall breakdown, shown in Figure 4.8. On the x-axis we have each configuration, and on the y-axis we have the share of what the stalls are caused by. Its clear that RS stalls is the culprit for all configurations, which means there are issues with the hardware that are causing the stalls. On our system, the RS

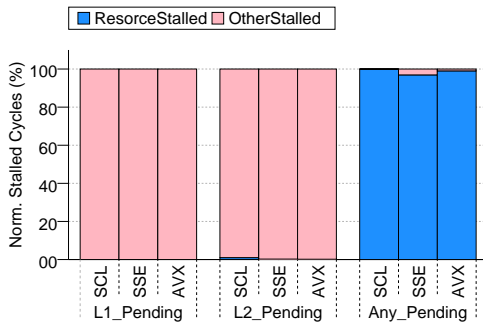


FIGURE 4.6: N-BODY NORMALIZED STALLED CYCLES (MEDIUM INPUT)

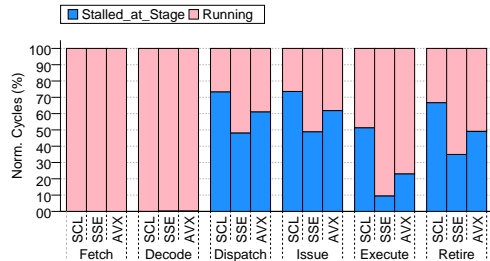


FIGURE 4.7: N-BODY NORMALIZED STALLS PER STAGE (MEDIUM INPUT)

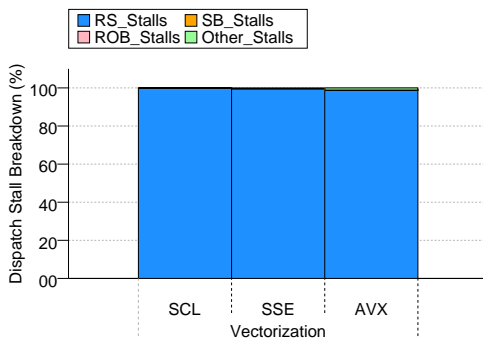


FIGURE 4.8: N-BODY DISPATCH STALLS BREAKDOWN (MEDIUM INPUT)

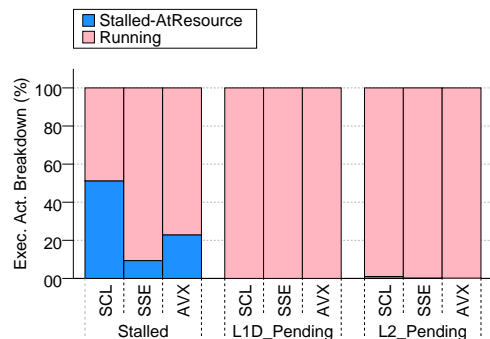


FIGURE 4.9: N-BODY EXECUTION ACTIVITY BREAKDOWN (MEDIUM INPUT)

supports 56 entries, which might be too small for this type of computation [Shimpi 2012, page 8].

Figure 4.9 graphs the rate of stalls when there were pending misses in the L1 and L2 cache levels. The x-axis show the rate of stalled/running cycles, together with the stall rate while there were pending accesses to the L1 and L2 cache levels, respectively. The y-axis show the percentage of the execution cycles that was stalled. We observe again, as we did in Figure 4.6, that any cache misses from accessing L1 and L2 are not the cause for the stalls.

### 4.2.3 Cache Performance

The L1 cache performance is shown in Figure 4.10 and Figure 4.11. On the x-axis, we have bars for each configuration in while reading from the L1 data cache, writing to the L1 data cache, and reading from the L1 instruction cache, respectively. On the y-axis, the figures show the number of or ratio of hits and misses to the total cache accesses for that configuration. We observe no issues with the L1 cache for all configurations: the miss rate is very low and should not cause any issues. The exception is L1I, but as we detailed in Section 3.4.3, this is caused by UOP fusion that is performed by the system.

Figure 4.12 and Figure 4.13 show the same data as the two previous graphs, but

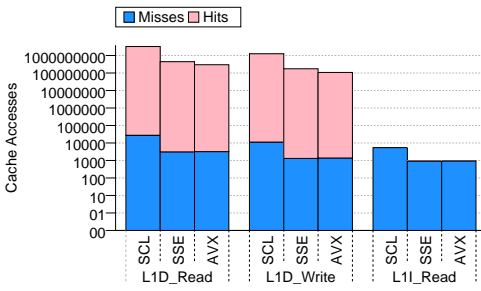


FIGURE 4.10: N-BODY L1 ACCESSES (MEDIUM INPUT, LOG SCALE)

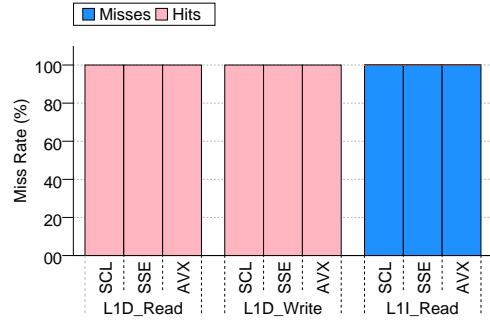


FIGURE 4.11: N-BODY L1 MISS RATE (MEDIUM INPUT)

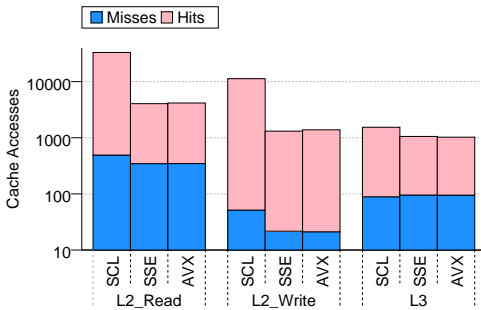


FIGURE 4.12: N-BODY L2 AND L3 ACCESSES (MEDIUM INPUT, LOG SCALE)

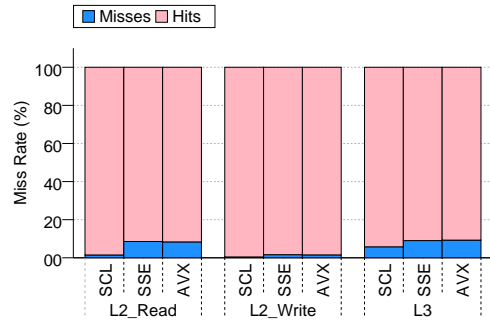


FIGURE 4.13: N-BODY L2 AND L3 MISS RATES (MEDIUM INPUT)

for the L2 and L3 cache levels instead. For L3 we do not show separate read and write graphs. Here, the miss rates are also low, although a bit higher than L1. Nevertheless, this should not cause big issues unless the L2/L3 cache miss latencies are high. The cache performance across all levels seem to indicate that this application is not memory-bound.

### 4.3 Potential for Further Performance Improvement

Compared to the single-precision performance of the original application, the improved `nbody` application performs better for the vectorized configurations: there are more retiring instructions, and the stalled cycles are significantly reduced. The AVX configuration is now also able to outperform the SSE configuration.

As the application most likely is CPU-bound, a clear next move would be to perform an experiment on newer hardware. According to [Tomás 2014], the Intel Skylake architecture has increased its number of RS entries to 97, which might decrease the amount of RS stalls and enable further speedup.

The critical operation in our particle computation, the vectorized reciprocal square root (`rsqrt`), exhibit far lower latencies on newer Intel architectures. [Intel n.d.(b)] show that on our Haswell system, SSE require 5 cycles latency for a throughput of 1, and AVX

requires 7 cycles latency for a throughput of 2. On the newer Skylake architecture, both configurations have been improved and now require 4 cycles latency for a throughput of 1. The AVX-512 SIMD extensions include a reciprocal square root intrinsic for both single- and double-precision values. *rsqrt14* calculates the inverse square root and truncates at 14 digits. According to Fog 2017b, running the single-precision operation on KNL requires 7 cycles latency for a throughput of 3. No latency/throughput is given for the double-precision. These instructions are available in the AVX-512F variation, supported on the Skylake-X/Cannonlake architectures for desktops and laptops, Skylake EP/EX for servers, and KNL. Another inverse square root intrinsic, *rsqrt28* (truncates at 28 digits), supports single- and double-precision values and is included in the AVX-512ER variation currently only available on KNL. [Intel n.d.(b)] show that the single-precision requires 8 cycles latency for a throughput of 3, and the double-precision requires 7 cycles latency for a throughput of 2.

# Chapter 5: Benchmark Analysis: N-queens

The 'n-queens' problem is a famous toy problem in chess which involves finding permutations in which an n number of queens can be placed on an n\*n chess board, such that no queen is vulnerable to an attack from any of the others. In practice, this means that no other queen can be placed on the same row, column, or diagonal as the last placed queen. One solution to this problem for n=8 is shown in Figure 5.1.

The problem dates back to 1848 when it consisted of placing 8 queens on a standard 8\*8 chess board. It was then called the '8-queens problem', and was studied by mathematicians such as Carl Friedrich Gauss [Bell et al. 2009]. Starting from 1869, the problem was generalized as the 'n-queens problem', and solutions for smaller values of n was found by working them out by hand. After the advent of modern computers, the n-queens problem has been rekindled by the ability to compute solutions for values of n higher than what is feasible to solve by hand. The problem is interesting from a computational viewpoint as there does not exist a closed-form solution, meaning that some computation is always necessary to find the solutions. The highest value of n where all solutions have been found is 27, an accomplishment that took over a year to compute on highly optimized hardware [Q27 n.d.]. There are also real-world scenarios that can be modeled as an n-queens problem, illustrating this application's usefulness other than as recreational mathematics [Bell et al. 2009].

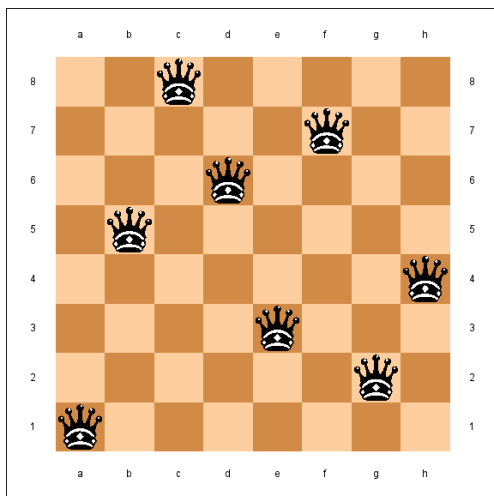


FIGURE 5.1: 8-QUEENS SOLUTION.  
SOURCE: [CHESSBOARD DRAWING N.D.]

TABLE 5.1: N-QUEENS SOLUTIONS UP  
TO N=22

N	Solutions	N	Solutions
1	1	12	14,200
2	0	13	73,712
3	0	14	365,596
4	2	15	2,279,184
5	10	16	14,772,512
6	4	17	95,815,104
7	40	18	666,090,624
8	92	19	4,968,057,848
9	352	20	39,029,188,884
10	724	21	314,666,222,712
11	2,680	22	2,691,008,701,644



## 5.1 Algorithm and Vectorization

While it is fairly trivial to place all queens arbitrarily such that the constraints are satisfied, the real challenge lies in identifying *all* solutions. However, using exhaustive search algorithms that permutes the whole board by placing a new queen after a valid placement has been found, the computations will quickly end up requiring vast amounts of time. For example, with  $n=8$  there are only 92 solutions, but  $\binom{64}{8} = 4,426,165,368$  possible placements. The exponential rise in solutions relative to problem size, shown in Table 5.1, makes this a highly computationally expensive task for large values of  $n$ . For this reason, various optimizations exist, e.g. rotating and reflecting a solution to produce additional ones. It is also possible to find solutions explicitly instead of using exhaustive search, which means the problem is solvable in polynomial time for all  $n$ 's. [Bernhardsson 1991]. Nevertheless, it is critical to carefully study *how* the solver performs the computation, as both help improve performance and portability to other systems. In addition to the solutions for  $n=27$ , which is over  $2.3 \times 10^{17}$ , the Q27 project yielded valuable insight that can be used for performance tuning, engineering and benchmarking purposes for similar systems [Q27 n.d.].

A typical  $n$ -queens solver application falls under the 'backtrack and branch-and-bound' dwarf, since it benefits from both of these methods: Solver applications tend to utilize heuristic search algorithms that reduce the available search space, e.g. depth-first search, and then backtracks to previously valid placements should the move be incorrect, discarding the unsuited regions in the search space while doing so. The computation is often performed recursively using a computation tree to structure the tasks.

[Ren et al. 2015] investigated how SIMD can be used to speed up the  $n$ -queens computation. They studied the recursive, task-parallel nature of the application, which was shown to limit SIMD utilization significantly because of varying rates of data input and high latencies of load/store operations. Based on this insight, they made transformations to the code that improve the data parallelism by creating a blocked recursive algorithm, collecting multiple tasks into a common data block that makes vectorization easier. They also made other transformations by optimizing how the stack operates, inserting vector intrinsics manually to improve branching performance, and using two novel techniques called re-expansion and stream compaction. Re-expansion is a scheduling policy system that fills and operates on the data blocks using a combination of breadth-first and depth-first search: the former to generate work until the block is full, and the latter to then execute work efficiently on the SIMD units. Figure 5.2 shows a computation tree after a partial breadth-first search. Dark nodes have already been explored, gray triangles are the rest of the tree, while the numbered nodes are threads that have been spawned to perform the current computations. To maximize thread utilization while facing leftwards, threads 2 and 3 execute work using depth-first search, while re-expansion allows threads 1 and 4 to toggle back to breadth-first search and generate more work in parallel. This technique enables their program to achieve high SIMD utilization at lower block sizes, allowing them to minimize cache bottlenecks usually associated with using large block sizes. For  $n$ -queens, the evaluation shows that their program achieves almost perfect SIMD utilization while still having a low rate of cache misses. Stream compaction is a block management scheme that structures similar types of tasks into the same blocks, improving SIMD utilization by reducing the need of CPU masking operations. However, the evaluation shows only a marginal performance improvement for  $n$ -queens, and stream compaction has thus been excluded from further analysis.

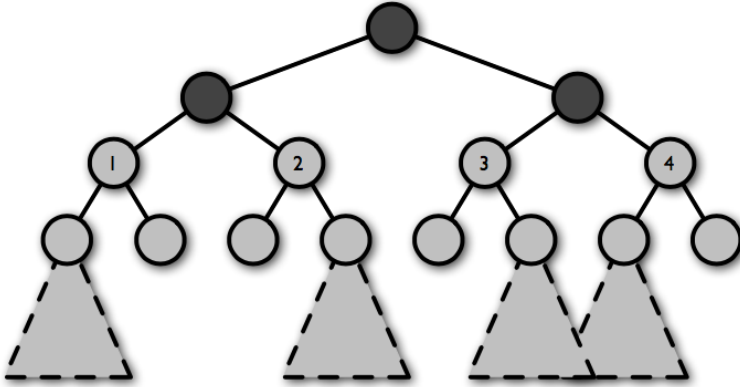


FIGURE 5.2: COMPUTATION TREE AFTER PARTIAL BREADTH-FIRST SEARCH. SOURCE: [REN ET AL. 2015]

We selected the `nqueens` application with re-expansion for evaluation. As well as a scalar version, it supports vectorization using SSE intrinsics. From comments in the code we learned that an AVX version was not made available at the time of implementation (2013), since it required hardware that had not been released yet. We have thus been able to analyze the program for both SSE and AVX configurations, as well as the scalar version that was used as a baseline. Ren et al. report that the data block size that achieved highest SIMD utilization for `nqueens` with re-expansion was  $2^{15}$ . We considered another block size,  $2^{18}$ , for AVX, since it gave a slightly faster runtime during an initial performance evaluation of several block sizes. However, further evaluation showed that it performed worse in all aspects compared to  $2^{15}$ . We have thus utilized the recommended block size for all our evaluation.

## 5.2 Results and Discussion

The `nqueens` application scales well between scalar and SSE, but fails to achieve a similar scale rate between SSE and AVX. The difference in input, shown in Section 1.2, indicates that this application is not input sensitive.

Both vectorized configurations have a significant amount of backend stalls; we can pinpoint some of these stalls to misses in the L1 cache. In addition, all configurations suffer from RS stalls during dispatch, suggesting that our hardware resources, which include a 56 entry RS, are not capable enough to handle this type of computation efficiently. Another explanation might be that the current implementation is not optimized for AVX, since it was not officially supported.

The data is analyzed and discussed in greater detail below.

### 5.2.1 Runtime, Cycle and Instruction Count

Figure 5.3 shows the runtime in seconds for the three configurations. From the similar slopes of the three curves we can infer that the runtime scales well between the different

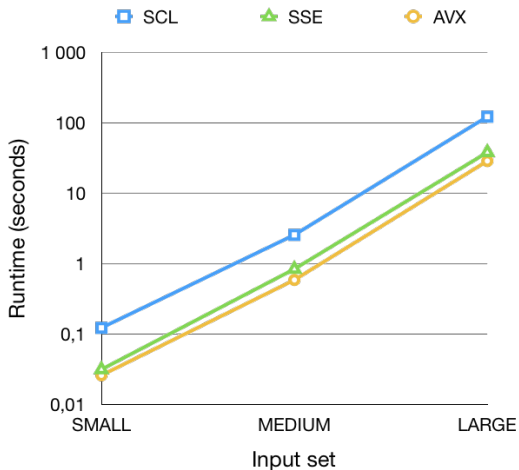


FIGURE 5.3: N-QUEENS RUNNING TIMES FOR THREE INPUT SIZES (LOG SCALE)

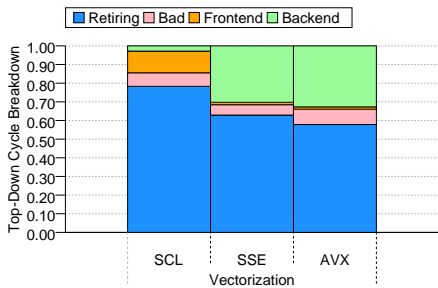


FIGURE 5.5: N-QUEENS TOP-DOWN CYCLE BREAKDOWN (MEDIUM INPUT)

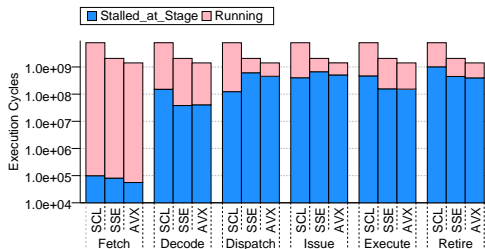


FIGURE 5.4: N-QUEENS EXECUTION CYCLES (MEDIUM INPUT, LOG SCALE)

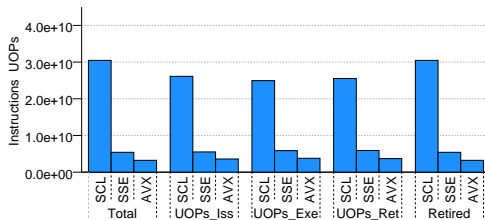


FIGURE 5.6: N-QUEENS INSTRUCTIONS PER STAGE (MEDIUM INPUT)

inputs. We also observe that the SSE configuration offers a substantial speedup over the scalar one, but that the AVX configuration is not able to offer similar speedup. A reason might be that there are stalls that limits it.

Figure 5.4 shows the total execution cycles required for each stage in the pipeline. We can see there are fewer decreased cycles between SSE and AVX than there are between scalar and SSE. We also observe that there is a noticeable amount of stalls during the dispatch, issue and retire stages. There is an almost equal number of stalled cycles for both the vectorized configurations, even though AVX requires less cycles in total. This might mean that there are some issues with the SSE configuration that becomes more prominent when using AVX.

Figure 5.5 shows a breakdown of the total instruction cycles for all three configurations using the top-down model. We gather that the vectorized configurations have about the same rate of backend stalling and retiring cycles, with AVX having a slightly higher rate of bad speculations. We see that the scalar configuration have more frontend stalls than the vectorized ones, which is logical since it needs to spend more time loading instructions. The vectorized configurations trade these stalls for backend ones. This probably means that they process more data than the memory and processor can keep

up with.

The total number of instructions and micro-operations (UOPs) are shown in Figure 5.6. We observe that there is a sizable reduction of instructions from scalar to SSE, and roughly half of these instructions again are required for the AVX configuration. The amount of reduced SSE instructions seems to large to be explained only by the increased vector length. Between SSE and AVX, the reduction indicates that the vectorization seems to be working as expected.

## 5.2.2 Stalls

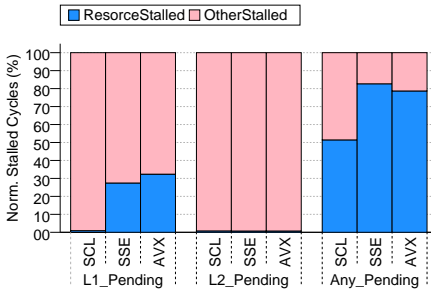


FIGURE 5.7: N-QUEENS NORMALIZED STALLED CYCLES (MEDIUM INPUT)

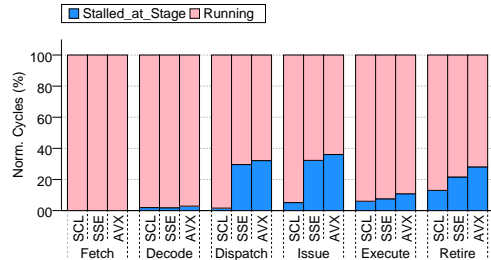


FIGURE 5.8: N-QUEENS NORMALIZED STALLS PER STAGE (MEDIUM INPUT)

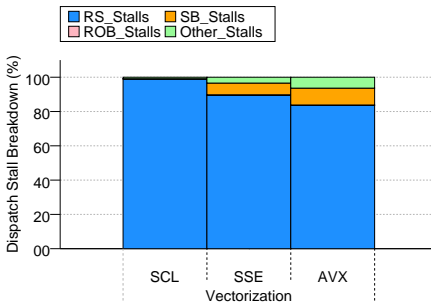


FIGURE 5.9: N-QUEENS DISPATCH STALLS BREAKDOWN (MEDIUM INPUT)

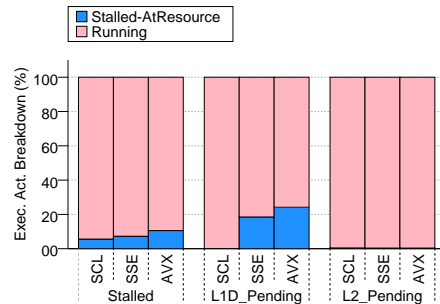


FIGURE 5.10: N-QUEENS EXECUTION ACTIVITY BREAKDOWN (MEDIUM INPUT)

From Figure 5.7 we can view the share of stalls when there were pending cache accesses. We see that about half the stalls for the vectorized configurations that occur with pending cache misses on all levels, happen when there are pending misses in L1. Pending L2 cache misses does not seem to contribute to these stalls. This suggests that the vectorized configurations have poorer cache performance than the scalar one, especially for L1. An explanation might be that they execute too fast for the data to reach the cache.

Figure 5.8 shows the rate of stalled and running cycles for each pipeline stage. As in Figure 5.4, we see that the share of stalls are highest in the dispatch, issue and retire stages, which reach up to 40% of the total cycles for the SSE and AVX configurations. It can be observed that the stall rate falls during the execution stage, which can infer that there is not an issue with the execution units being used inefficiently. This leaves poor

memory performance as a feasible explanation.

The stall breakdown in the dispatch stage is shown in Figure 5.9. We notice that the largest share of stalls for all configurations are RS stalls, which might be caused by a lack of computational resources. [Shimpi 2012, page 8] show that the RS in our system supports 56 entries; the RS stall share seems to indicate that this number is too small for this type of computation. For the vectorized configurations, we notice that RS stalls are gradually traded for SB stalls and other stalls. This trend becomes more prominent as we increase vector length. SB stalls are caused by the memory subsystem not being able to store the generated amount of data in time. The increasing amounts of these stalls indicates that the vectorized configurations execute too fast and has to wait in order to store the data back in memory.

Figure 5.10 graphs the rate of stalls when there were pending misses in the L1 and L2 cache levels. For the vectorized configurations, we notice a prominent rate of stalled cycles when there were pending misses in L1D. This can be used to further support the claim that the vectorized applications have poor L1 cache performance. From Figure 5.9 we notice that for the vectorized configurations, the stalled cycles when there were L1D pending misses match with the increase of SB and other stalls.

### 5.2.3 Cache Performance

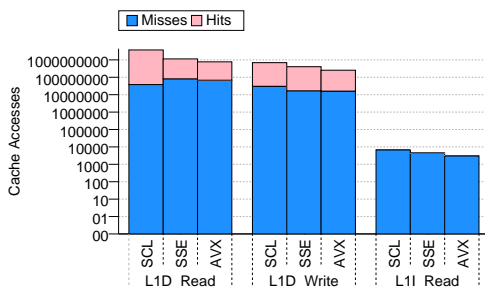


FIGURE 5.11: N-QUEENS L1 ACCESSSES (MEDIUM INPUT, LOG SCALE)

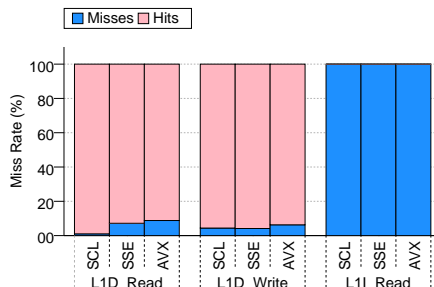


FIGURE 5.12: N-QUEENS L1 MISS RATE (MEDIUM INPUT)

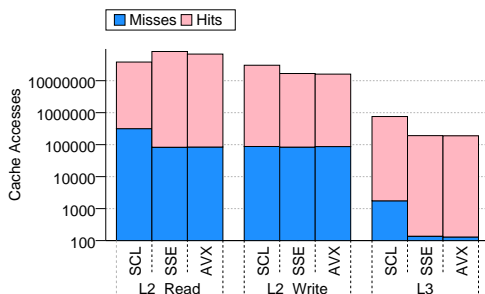


FIGURE 5.13: N-QUEENS L2 AND L3 ACCESSSES (MEDIUM INPUT, LOG SCALE)

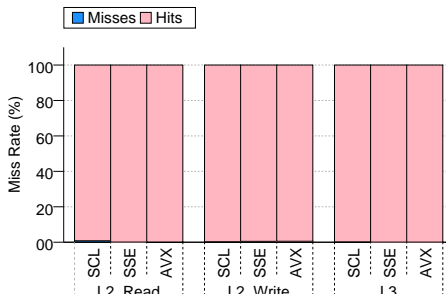


FIGURE 5.14: N-QUEENS L2 AND L3 MISS RATES (MEDIUM INPUT)

The L1 cache performance is shown in Figure 5.11 and Figure 5.12. We observe that the miss rate in L1 for the vectorized configurations is around 10%. This should not be able

to explain the increased stalls unless the cache miss latency of L1 is monumentally high, which it is not. [Shimpi 2012, page 9] show that on our system, L1 latency is 4 cycles, while L2 latency is 11 cycles. For L1I we have a 100% miss rate, which can be explained by UOP fusion.

Figure 5.13 and Figure 5.14 show the same data as the two previous graphs, but for the L2 and L3 cache levels instead. We observe minuscule miss rates across all configurations.

### 5.3 Potential for Further Performance Improvement

There seems to be two main explanations for the bottlenecks experienced during the evaluation: either the hardware resources are not sufficient enough to perform the computations efficiently, or there is a need for algorithm optimizations.

The first, like in `nbody`, can be remedied by running experiments on newer hardware, e.g. Skylake which has increased its RS size to 97 entries. In addition, the instructions used for SSE and AVX show very little variance in latency and throughput: most intrinsics have latencies of one cycle, which have not been further reduced on newer architectures.

The non-linear reduction in instructions seem to indicate that the algorithm needs some reworking to benefit from AVX, which also suffer from higher rates of bad speculations. What might sound reasonable in order to mitigate this problem is increasing the block size, which will fit more values and thus increase AVX performance. However, this will most likely be counterproductive and can result in performance degradation if we increase it past a certain point. [Ren et al. 2015] explain: "Larger block sizes lead to more work that can be vectorized, increasing SIMD utilization. However, large blocks suffer from poor locality, increasing cache misses. To achieve good performance, therefore, we want to achieve good SIMD utilization with the smallest possible block size."

# Chapter 6: Benchmark Analysis: Needleman-Wunsch and Smith-Waterman

Sequence alignment is a field in bioinformatics for aligning and comparing sequences of biological data or *residues*, e.g. nucleic acids or proteins. Its purpose is to uncover regions of similarity, which may point to a relationship between the two sequences. This, in turn, can be used as the basis in the development of new bio-medical products.

The alignment process is not as simple as shifting one sequence until you find a segment with the most matches. An efficient alignment algorithm also has to consider similarities between the nucleic acids or proteins, and be able to insert gaps if one of the sequences have some extra information in between an otherwise matching region. It is also worth noting that an alignment is only one of many possibilities, and that the calculated one therefore may not be optimal [Al-Karadaghi n.d.].

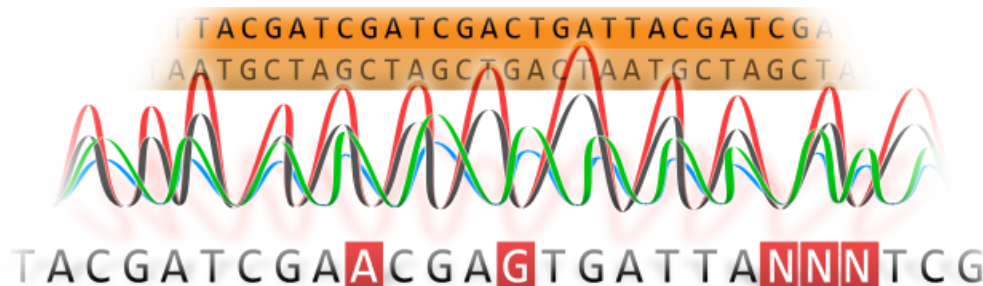


FIGURE 6.1: SEQUENCE ALIGNMENT. SOURCE: SEQUENCE ALIGNMENT ILLUSTRATION N.D.

We typically choose between global or local alignment algorithms, or a hybrid of the two. Global alignment means that we consider the two full sequences and try to find the longest matching sequence between the two. This is most efficient when the sequences are similar in size and structure. The Needleman-Wunsch (NW) algorithm is an example of a global sequence alignment algorithm. Local alignment consider regions of interest, often called a query sequence, and aligns them to similar regions inside a longer sequence, called the database sequence. A short query sequence and a longer database sequence generally gives the best results, and it is preferable that the two sequences are of relative similarity. The Smith-Waterman (SWat) algorithm, a variation of NW, is an algorithm used for local sequence alignment.

## 6.1 Algorithm and Vectorization

Both SWat and NW represent the two sequences as axes in a scoring matrix, and utilize dynamic programming techniques by computing a score between matrix element pairs

and storing the direction to travel between them. The scores are determined by a scoring system which decides how many points each comparison should be awarded. Later, the optimal aligned sequence is calculated as an optimal path between two points in the matrix based on these scores and directions. Both algorithms are frequently used in the industry since they will find or at least determine if the optimal alignment is available, given that the optimal scoring system is used [Roberts n.d.].

Research shows that SIMD can be used to offer significant speedups to both the NW and SWat algorithms. [Fakirah et al. 2015] evaluated a GPU implementation of NW and reported great speedup. [Rognes and Seeberg 2000] present a novel approach to vectorizing SWat using MMX and SSE instructions, where they store 8 subsequent residues in the 64 bit MMX vectors, and are thus able to perform 8 concurrent operations in parallel, gaining notable speedup. [Farrar 2006] use a 'striped' approach, meaning that the parallelization is performed at different parts of the query sequence to reduce data dependency latencies. The result is 16 simultaneous operations using SSE instructions. [Rognes 2011] improve on this implementation by presenting SWIPE, which is able to compare a query residue to 16 different database residues in parallel using SSE3 instructions.

For this thesis, we have analyzed Opal [Šošić n.d.], a project which extends SWIPE with support for both SSE4.1 and AVX2, as well as supporting three other alignment modes in addition to Swat, including NW. By using AVX, we are able to process twice the number of residues, i.e. 32, in parallel. The application does not include a scalar version. Thus, to get a baseline performance, we halved the available vector length width in the SSE version's load/store instructions, so that the rest of the operations in practice only utilized 64 of the available 128 bits in the vector.

## 6.2 NW: Results and Discussion

The results show that all configurations suffer from large backend stalls. While the scalar and AVX configurations behave as expected, the SSE experiences a bottleneck that slows down overall performance. The results from other input sizes, shown in Section 1.3, indicate that this application is not input sensitive. The backend stalls seem to originate from register spilling, i.e. running out of physical registers, which hits the SSE configuration harder than it does for AVX. While both utilizing 16 registers, AVX require fewer because it fits twice the data inside the same vector, compared to SSE. We see relatively few misses in L1, and some slight ones for L2 and L3. The cache misses do not indicate to cause any stalls, but rather that the program processes data faster than the cache can handle. Thus, it seems this application is not memory bound.

The data is analyzed and discussed in greater detail below.

### 6.2.1 Runtime, Cycle and Instruction Count

Figure 6.2 shows the runtime in seconds for the three configurations. We observe from the similar slopes of the three curves that this application scales in runtime for the different inputs. The SSE configuration provides a reasonable speedup from the scalar one. However, the AVX configuration offers an even greater speedup than that. A possible explanation is that the AVX configuration is highly optimized, or there are stalls in the SSE configuration that slows it down.

Figure 6.3 shows the total execution cycles required for each stage in the pipeline. We notice the same trend in the amounts of cycles as was found for runtime, which is that



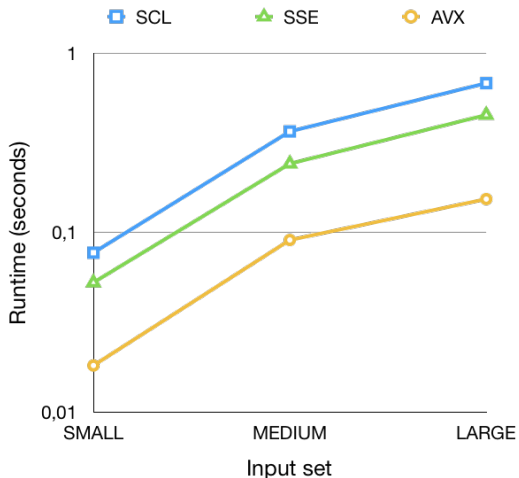


FIGURE 6.2: NW RUNNING TIMES FOR THREE INPUTS (LOG SCALE)

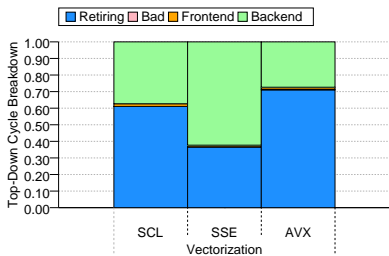


FIGURE 6.4: NW TOP-DOWN CYCLE BREAKDOWN (MEDIUM INPUT)

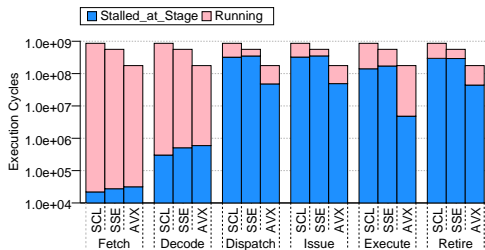


FIGURE 6.3: NW EXECUTION CYCLES (MEDIUM INPUT, LOG SCALE)

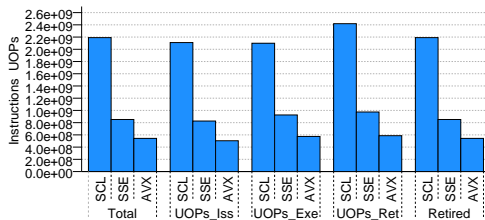


FIGURE 6.5: NW INSTRUCTIONS PER STAGE (MEDIUM INPUT)

AVX offers a greater reduction from SSE than what SSE manages from scalar. We also notice that there is a substantial amount of stalls during the dispatch, issue and retire stages. The stalls drop during execution for AVX, but not for SSE; this suggests that SSE struggles utilizing the execution units efficiently, while AVX does not.

Figure 6.4 shows a breakdown of the total instruction cycles for all three configurations using the top-down model. We observe that the cycles for all configurations either retires or stalls during backend. While AVX offers slightly more retired cycles than scalar, SSE manages only to retire about half of these due to increased backend stalls. From the top-down model, the cause of the SSE bottleneck is likely that it waits for the memory subsystem to complete, or that there is a combination of high instruction latencies and poor utilization of hardware resources.

The total number of instructions and micro-operations (UOPs) are shown in Figure 6.5. The instructions seem to scale as we would expect from the increased vector length.

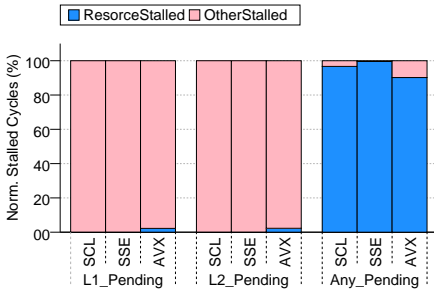


FIGURE 6.6: NW NORMALIZED STALLED CYCLES (MEDIUM INPUT)

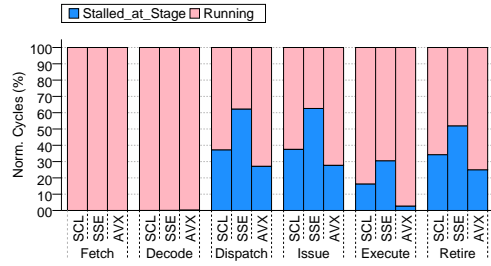


FIGURE 6.7: NW NORMALIZED STALLS PER STAGE (MEDIUM INPUT)

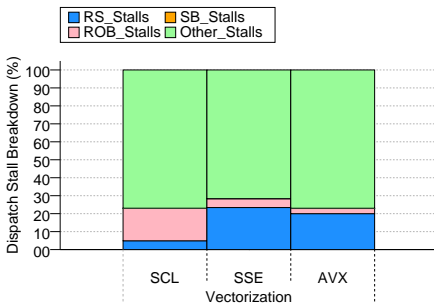


FIGURE 6.8: NW DISPATCH STALLS BREAKDOWN (MEDIUM INPUT)

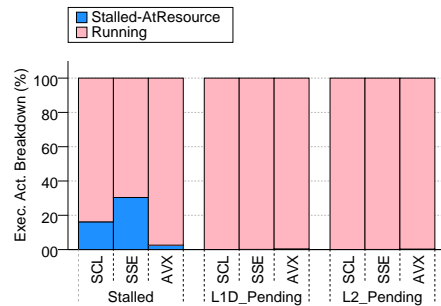


FIGURE 6.9: NW EXECUTION ACTIVITY BREAKDOWN (MEDIUM INPUT)

## 6.2.2 Stalls

From Figure 6.6 we can view the share of stalls when there were pending cache accesses. None of the stalls that appear when there are pending misses are present when there are pending misses in L1 and L2, except for some slight ones in the AVX configuration. The high rate of stalls in Any\_Pending can likely be explained by the compulsory misses happening due to UOP fusion.

Figure 6.7 shows the rate of stalled and running cycles for each pipeline stage. As in Figure 6.3, we see that the share of stalls are highest in the dispatch, issue and retire stages, and the SSE configuration has a consistently higher stall rate than the other two. We also observe that there is a minuscule stall rate for the AVX configuration during execution, which indicates that it has been optimized to run efficiently.

The stall breakdown in the dispatch stage is shown in Figure 6.8. We observe that there are a large amount of other stalls across all configurations, which means that there is a lack of free execution units or physical registers. The second largest stall share for the scalar configuration is ROB stalls, which means there are some stalls while speculating. The second largest for the vectorized configurations is RS stalls, meaning there are stalls while renaming registers. The stall distribution is fairly similar for the vectorized configurations, so an explanation for the performance gap might be that it is more costly to perform these stalls for SSE. There is also the issue of register spilling: both SSE and AVX include 16 registers when running on a 64-bit system, but as the AVX registers store double the amount of data, this configuration requires less physical registers in total. This

means any issues due to lack of registers will be more prominent for SSE.

Figure 6.9 graphs the rate of stalls when there were pending misses in the L1 and L2 cache levels. There seem to be no stalls caused by misses in L1D or L2, suggesting that the SSE stalls must be caused by a lack of hardware resources instead.

### 6.2.3 Cache Performance

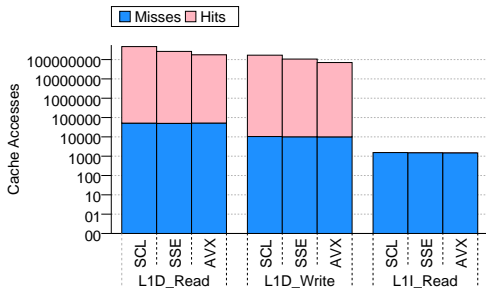


FIGURE 6.10: NW L1 ACCESSES (MEDIUM INPUT, LOG SCALE)

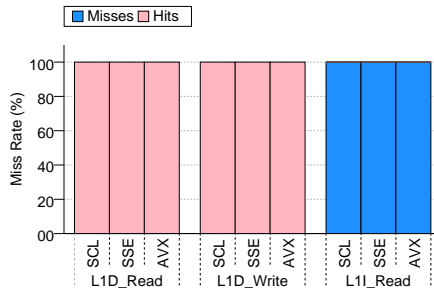


FIGURE 6.11: NW L1 MISS RATE (MEDIUM INPUT)

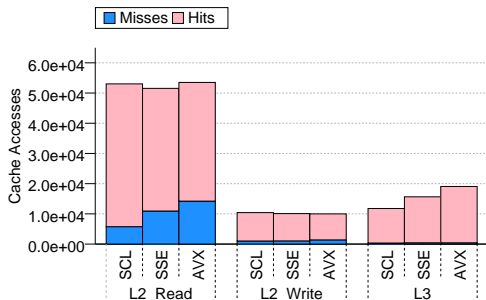


FIGURE 6.12: NW L2 AND L3 ACCESSES (MEDIUM INPUT, LOG SCALE)

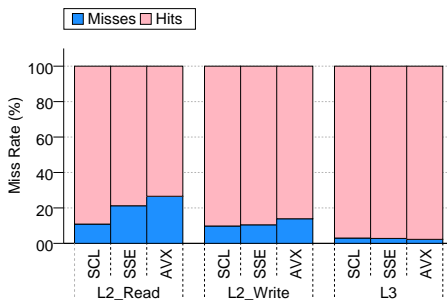


FIGURE 6.13: NW L2 AND L3 MISS RATES (MEDIUM INPUT)

The L1 cache performance is shown in Figure 6.10 and Figure 6.11. We observe that only minuscule miss rates for L1D, implying that this cache level functions efficiently. For L1I we have a 100% miss rate, which can be explained by UOP fusion.

Figure 6.12 and Figure 6.13 show the same data as the two previous graphs, but for the L2 and L3 cache levels instead. We notice rising miss rates in the L2 level as we increase vector length, as well as some slight, constant miss rates for L3. This can likely be explained by the configurations processing data faster than it manages to reach the cache. It also seems to pose no issues to overall efficiency, as the AVX configuration, with the highest miss rate, is the by far fastest configuration.

## 6.3 SWat: Results and Discussion

The results for `SWat` show that the application is behaving excellently. The runtime seems to scale linearly between the input sizes and configurations. Most cycles are retiring, but

there are some significant frontend and backend stalls, the latter which are more prominent for the SSE configuration. The stalls seem to be caused by register spilling, as was the case with NW. However, the effects seem to not be as severe this time. Cache performance is overall good, with low miss rates that does not indicate any connection to the stalls. This application does not seem to be memory bound as well.

The data is analyzed and discussed in greater detail below.

### 6.3.1 Runtime, Cycle and Instruction Count

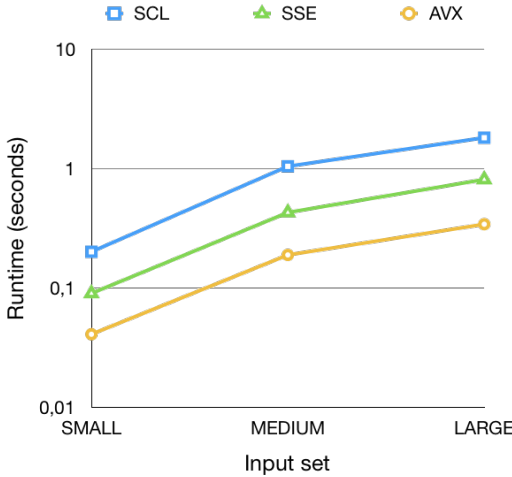


FIGURE 6.14: SWAT RUNNING TIMES FOR THREE INPUTS (LOG SCALE)

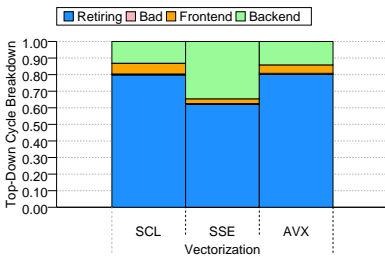


FIGURE 6.16: SWAT TOP-DOWN CYCLE BREAKDOWN (MEDIUM INPUT)

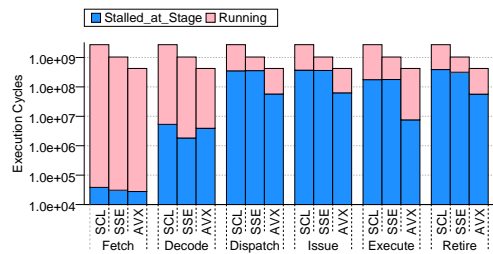


FIGURE 6.15: SWAT EXECUTION CYCLES (MEDIUM INPUT, LOG SCALE)

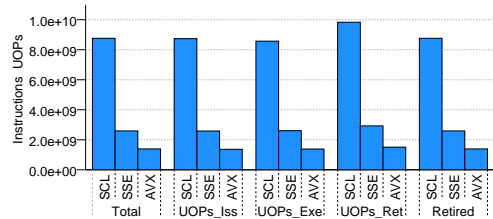


FIGURE 6.17: SWAT INSTRUCTIONS PER STAGE (MEDIUM INPUT)

The runtime in Figure 6.14 graphs the runtime in seconds for the three configurations. We observe that the runtime scales very well between the different inputs and vector sizes.

The same goes for execution cycles. Figure 6.15 shows the total execution cycles required for each stage in the pipeline. We notice a similar rate of decrease in execution cycles as we double the vector width. This data shows the different configurations are behaving as expected. However, there are substantially higher amounts of stalls in the SSE configuration than for AVX. We can also observe that the amounts of stalls in all configurations are significantly lower than for those in NW.

Figure 6.16 shows a breakdown of the total instruction cycles using the top-down model. From the data we can gather that most cycles finish successfully, although there are some backend stalls and bad speculations present. The worst performing configuration is SSE, which has three times the backend stalls as the other two. A likely reason might be register spilling, which was a possible explanation for the similar behavior in NW.

The total number of instructions and micro-operations (UOPs) are shown in Figure 6.17. We observe that the number of instructions decrease substantially between scalar and SSE, and then roughly halve between SSE and AVX. Overall, the instruction count seems to scale as we increase vector length.

### 6.3.2 Stalls

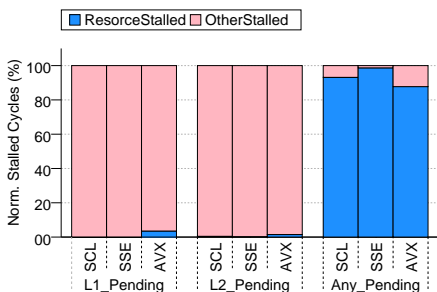


FIGURE 6.18: SWAT NORMALIZED STALLED CYCLES (MEDIUM INPUT)

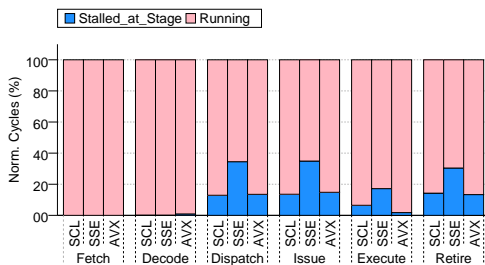


FIGURE 6.19: SWAT NORMALIZED STALLS PER STAGE (MEDIUM INPUT)

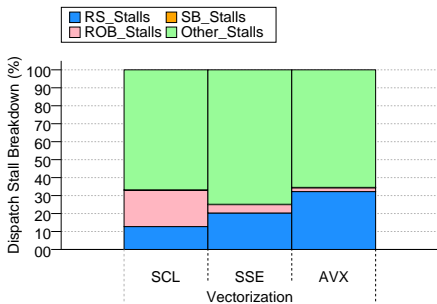


FIGURE 6.20: SWAT DISPATCH STALLS BREAKDOWN (MEDIUM INPUT)

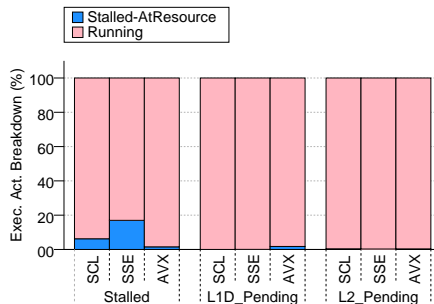


FIGURE 6.21: SWAT EXECUTION ACTIVITY BREAKDOWN (MEDIUM INPUT)

From Figure 6.18 we can view the rate of stalls when there were pending cache accesses. We can see that no stalls in the L1 and L2 cache levels for the scalar and SSE configurations are caused by cache misses, but a small share of the AVX stalls are. Previously, we saw that AVX both had significantly fewer total and stalled execution cycles than the other configurations, so it is likely that any stalls here will more easily surface.

Next, we investigate whether the stalls are constant or if they rise during a particular stage, by looking at Figure 6.19. We know that the stalls start to propagate from the dispatch stage and outwards, but fall substantially during execution. It can be observed that the SSE configuration has the highest rate of stalls in all stages, while scalar and

AVX have about the same stall rate in these stages. This application has slightly higher rate of stalled cycles for the AVX configuration than what was present in NW. Nevertheless, the **SWat** stall rate reaches slightly above 30%, while for NW it reached over 60%.

The stall breakdown in the dispatch stage is shown in Figure 6.20. We see that there is a slight increase in other stalls for the SSE configuration compared to the other two. As was the case for NW, other stalls indicate that there are a lack of physical or execution units. However, the SSE stalls surpass the amount from NW. If there are similar bottlenecks present in both applications, we can infer that the SSE stalls in **SWat** are also caused by register spilling, but cause further issues for **SWat**.

Figure 6.21 compares the total stalled execution cycles to the cycles that included pending misses to L1D or L2. We notice that the few stalls present in the AVX configuration can be explained by L1 misses, likely due to it processing data faster than the L1 can handle. However, as SSE does not exhibit any L1 misses, it implies it is not a source of performance degradation. It seems the SSE stalls are caused by something else.

### 6.3.3 Cache Performance

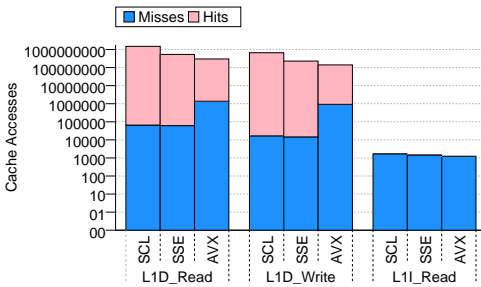


FIGURE 6.22: SWAT L1 ACCESSES (MEDIUM INPUT, LOG SCALE)

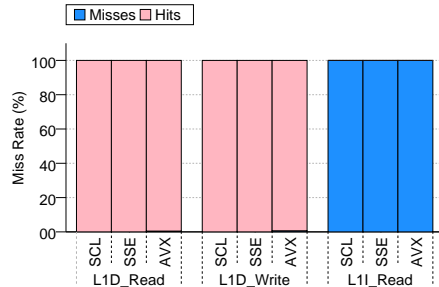


FIGURE 6.23: SWAT L1 MISS RATE (MEDIUM INPUT)

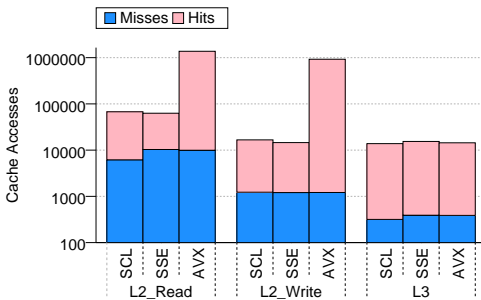


FIGURE 6.24: SWAT L2 AND L3 ACCESSES (MEDIUM INPUT, LOG SCALE)

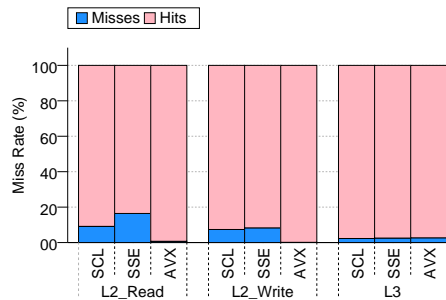


FIGURE 6.25: SWAT L2 AND L3 MISS RATES (MEDIUM INPUT)

The L1 cache performance is shown in Figure 6.22 and Figure 6.23. We observe from these two figures that we have a relatively small amount of cache misses in the data cache. The misses in the instruction cache is likely caused by UOP fusion. The AVX configuration exhibits 100 times the number of data cache misses than the other two configurations; however, they seem to be out of the critical path of the application since the running

times perform as expected. That is, despite the higher miss rate, data is still provided in time.

Figure 6.24 and Figure 6.25 show the same data as the two previous graphs, but for the L2 and L3 cache levels instead. We can see that the AVX has an 100 times increase in L2 accesses, which is expected given this is the same rate of increased misses in L1. Other than that, the accesses are quite similar across all configurations. While the AVX configuration has more L2 accesses, it has approximately the same misses as the other two, resulting in minuscule miss rates.

## 6.4 Potential for Further Performance Improvement

Overall, the performance of these two benchmarks is more than acceptable. For further speedup, it would seem that the only practical solution is to rework the algorithm in order to mitigate the register spilling issues. This can be done by either using a compiler with better register allocation, or writing our own register allocation routine directly in assembly and bypassing the built-in compiler function so we can control exactly which registers are used when.

# Chapter 7: Conclusion and Future Work

The lack of a SIMD-aware benchmarking tool that fully covers the Berkeley dwarfs taxonomy led Cebrian et al. to initiate SIMDwarfs. In this thesis, we have contributed to this project by analyzing the performance and scalability of four SIMD capable applications, `nbody`, `nqueens`, `NW`, and `SWat`, covering three previously uncovered dwarfs. We have ported all applications to the ParVec wrapper library, enabling us to evaluate configurations using SSE and AVX SIMD extensions, as well as a scalar configuration for baseline.

All mandatory research tasks have been carried out. T1, which was to detail how vectorization is applied, has been covered in the first sections of the benchmark analysis chapters. T2, determine benchmark performance and scalability, was carried out by the data analysis, which show that all applications experience stalls due to lack of hardware resources, i.e. they are CPU-bound. Architectural bottlenecks were located (T3): `nbody` and `nqueens` have high amounts of RS stalls, implying the 56 entries on our evaluation system (Haswell) is not large enough to support this type of computation efficiently, especially for AVX. `NW` and `SWat` perform well, but show significant amounts of 'other' stalls likely explained by register spilling, i.e. the compiler runs out of logical registers. This is more noticeable in the SSE configuration, as the vectors hold less data than AVX. Since these mitigation strategies (T3.1) involve upgrading the hardware capabilities or reworking the algorithm used, T3.2 and T3.3 are out of reach for this thesis. Finally, T4 has been performed as the vectorized applications can all be found in the SIMDwarfs (private) github repo.

## 7.1 Future Work: Towards Complete Coverage

With the three additional dwarfs covered through this thesis, SIMDwarfs now covers 10 out of 13 dwarfs. The three dwarfs that remain until we achieve full coverage is combinational logic, graphical models and finite state machines. Due to time constraints, we have not analyzed these in detail. In order to aid future SIMDwarfs developments, this section combines what was found during the previous literature survey and new insight that was gained during this thesis.

### 7.1.1 Combinational Logic

One benchmark, `CRC`, is considered for this dwarf. This is a Cyclic redundancy check application, which is used for code error detection. The benchmark included in [OpenDwarfs n.d.(b)] works by computing a CRC32 value using the "slice-by-8" algorithm which performs XOR operations on 64 bits of data at a time [Kounavis et al. 2005]. SSE 4.2 includes intrinsics for calculating CRC32 values in hardware, but no AVX equivalent is currently available. [Suresh n.d.] provides an application that utilizes these SSE intrinsics to significantly speed up computations. This dwarf was only recently discovered not to be covered, and has therefore not been further analyzed.



### 7.1.2 Graphical Models

One benchmark, `HMM`, is considered for this dwarf as well. The benchmark included in [OpenDwarfs n.d.(c)] is an implementation of the Baum-Welch algorithm, which is used to find the unknown parameters of a hidden markov model. `HMMlib` from [Sand n.d.] includes support for SSE SIMD extensions. The authors found that using their library on modern processors provided significant speedups for most Baum-Welch implementations. [Huo et al. n.d.] provide a vectorized `hmm` implementation using AVX2 intrinsics. They report a 5.70X speedup over baseline for a single threaded Baum-Welch algorithm using AVX. We did not manage to run these applications due to problems getting support libraries installed on our system, and thus no further analysis has been performed.

### 7.1.3 Finite State Machines

From the multiple benchmarks that are considered for this dwarf, we managed to find two that included a vectorized implementation. The first is `x264`, a video compression application based on the H.264/AVC video encoder. A vectorized implementation, using SSE SIMD extensions, is provided by [Cisco n.d.]. The second is `dedup`, a 'deduplication' data compression tool. A vectorized implementation also using SSE SIMD extensions is provided by [Ghosh n.d.]. Due to some trouble getting required support libraries installed on our system, we have not further analyzed either of these applications.

## 7.2 The Future of SIMD Extensions

Intel is currently adding support for AVX-512, which are 512-bit extensions to the AVX SIMD extensions, to its newest architectures. AVX-512 includes multiple subsets, which will be supported on different, specialized hardware, although basic operations from the AVX-512F (foundation) subset requires support on all hardware. In addition to increased vector length, it supports up to 32 vector registers on 64-bit systems. There are also further improvements, e.g. new mask registers that supports strided accesses and conditional operations, and a new encoding format (EVEX) that supports up to four operands [Wikipedia 2017]. The new SIMD extensions are currently supported on the KNL (2016) and Skylake-X (2017) architectures, as well as the upcoming Intel architectures for desktops and servers, codenamed Cannonlake and Skylake Xeon Purley, and a new Xeon Phi product codenamed Knights Mill, all expected to be released in 2017. No other manufacturer using the x86 architecture have yet announced support for AVX-512. [Fog 2013] notes that there is room for further vector expansions up to 1024 bits, which indicates that such a technology can emerge in the future.

Scalable vector extension (SVE), a collaboration between Fujitsu and ARM, was announced in 2016. This SIMD extension technology was designed to complement ARM's previous NEON extensions in order to further improve vectorization of HPC scientific workloads. Feature-wise, SVE resembles a vector architecture. It introduces multiple improvements over NEON, including increased flexibility with regards to vector length, ranging from 128 to 2048 bits, support for gather-scatter operations and a programming model that adapts to the available vector length, as well as improved auto-vectorization capabilities. The full list of improvements is found in [Stephens 2016]. The instruction set was released publicly in 2017.

There have also been made efforts to speed up web browsing by utilizing SIMD, e.g.

the Dart programming language [McCutchan 2013]. They claim a significant speedup in Javascript performance for activities such as 3D graphics, image/audio processing and numeric computations, compared to not using SIMD hardware acceleration. A 3D animation written in Dart and running in a web browser was able to go from 34 to 126 animated figures while keeping a frame rate of 60 frames per second, by turning on support for SIMD [Lund et al. 2013, 49m52s].

As SIMD-aware benchmarking tools such as SIMDwarfs become available, who knows what the future of SIMD will hold?

# Bibliography

## Books

- Patterson, David A. and Hennessy, John L. (2014). *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. Elsevier. ISBN: 9780124077263.
- Stallings, William (2013). *Computer Organization and Architecture*. 9th ed. Pearson. ISBN: 9780132936330.
- National Research Council and others (2005). *Getting up to speed: The future of super-computing*. National Academies Press.
- Hennessy, John L. and Patterson, David A. (2012). *Computer Architecture: A Quantitative Approach*. 5th ed. Elsevier. ISBN: 9780123838728.

## Theses and Dissertations

- Bienia, Christian (2011). “Benchmarking modern multiprocessors”. PhD thesis. Princeton University.

## Articles

- Flynn, Michael J (1972). “Some computer organizations and their effectiveness”. In: *IEEE transactions on computers* 100.9, pp. 948–960.
- Lee, Victor W et al. (2010). “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News* 38.3, pp. 451–460.
- Cebrian, Juan M, Jahre, Magnus, and Natvig, Lasse (2015). “ParVec: vectorizing the PARSEC benchmark suite”. In: *Computing. Archives for Informatics and Numerical Computation* 97.11, p. 1077.
- Smith, James E, Faanes, Greg, and Sugumar, Rabin (2000). “Vector instruction set support for conditional operations”. In: 28.2.
- Weicker, Reinhold P (1990). “An overview of common benchmarks”. In: *Computer* 23.12, pp. 65–75.
- Rabaey, Jan M et al. (2008). “Workloads of the Future”. In: *IEEE Design & Test of Computers* 25.4.
- Dubey, Pradeep (2005). “Recognition, mining and synthesis moves computers to the era of tera”. In: *Technology@ Intel Magazine* 9.2, pp. 1–10.
- Williams, Samuel, Waterman, Andrew, and Patterson, David (2009). “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4, pp. 65–76.
- Blem, Emily, Sinclair, Matthew, and Sankaralingam, Karthikeyan (2011). “Challenge benchmarks that must be conquered to sustain the GPU revolution”. In: *CELL* 1024.8, p. 228.

- Bell, Jordan and Stevens, Brett (2009). “A survey of known results and research areas for n-queens”. In: *Discrete Mathematics* 309.1, pp. 1–31.
- Bernhardsson, Bo (1991). “Explicit solutions to the N-queens problem for all N”. In: *ACM SIGART Bulletin* 2.2, p. 7.
- Rognes, Torbjørn and Seeberg, Erling (2000). “Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors”. In: *Bioinformatics* 16.8, pp. 699–706.
- Farrar, Michael (2006). “Striped Smith–Waterman speeds database searches six times over other SIMD implementations”. In: *Bioinformatics* 23.2, pp. 156–161.
- Rognes, Torbjørn (2011). “Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation”. In: *BMC bioinformatics* 12.1, p. 221.

## Proceedings

- Cebrian, Juan M, Jahre, Magnus, and Natvig, Lasse (2014). “Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks”. In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, pp. 66–75.
- Esmaeilzadeh, Hadi et al. (2011). “Dark silicon and the end of multicore scaling”. In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 3. ACM, pp. 365–376.
- Cebrian, Juan M, Natvig, Lasse, and Meyer, Jan Christian (2012). “Improving energy efficiency through parallelization and vectorization on intel core i5 and i7 processors”. In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion: IEEE*, pp. 675–684.
- Maleki, Saeed et al. (2011). “An evaluation of vectorizing compilers”. In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, pp. 372–382.
- Rajovic, Nikola, Rico, Alejandro, Vipond, James, et al. (2013). “Experiences with mobile processors for energy efficient HPC”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, pp. 464–468.
- Rajovic, Nikola, Rico, Alejandro, Mantovani, Filippo, et al. (2016). “The mont-blanc prototype: an alternative approach for HPC systems”. In: *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, pp. 444–455.
- Ren, Bin et al. (2015). “Efficient execution of recursive programs on commodity vector hardware”. In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM, pp. 509–520.
- Yasin, Ahmad (2014). “A top-down method for performance analysis and counters architecture”. In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, pp. 35–44.
- Fakirah, Maged et al. (2015). “Accelerating needleman-wunsch global alignment algorithm with gpus”. In: *Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of*. IEEE, pp. 1–5.
- Kounavis, Michael E and Berry, Frank L (2005). “A systematic approach to building high performance software-based CRC generators”. In: *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*. IEEE, pp. 855–862.

## Technical Reports

- Asanovic, Krste et al. (2006). *The landscape of parallel computing research: A view from Berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
- Fog, Agner (2017a). *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. Tech. rep. Technical University of Denmark.
- (2017b). *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Tech. rep. Technical University of Denmark.

## Unpublished Documents

- De Frène, Christian (2016). “Towards a Vectorized Benchmark Suite Covering the Berkeley Dwarfs”. Specialization project at NTNU.
- Colella, Phil (2004). “Defining Software Requirements for Scientific Computing”. Presentation.

## Online Documents and Resources

- Stringer, Lynd (2016). *Vectors: How the Old Became New Again in Supercomputing*. Accessed: 2017-07-07. URL: <https://www.hpcwire.com/2016/09/26/vectors-old-became-new-supercomputing/>.
- PARSEC (n.d.). *The PARSEC Benchmark Suite*. Accessed: 2017-07-10. URL: <http://parsec.cs.princeton.edu>.
- Pozzi, Laura and Silvano, Cristina (2012). *Data-Level Parallelism in SIMD and Vector Architectures*. Accessed: 2017-06-19. URL: <http://home.deib.polimi.it/silvano/FilePDF/AAC/Lesson13A-SIMD-vector.pdf>.
- Koopman, Philip (1998). *Vector architecture*. Carnegie Mellon University. URL: [https://www.ece.cmu.edu/~ece548/handouts/16v\\_arch.pdf](https://www.ece.cmu.edu/~ece548/handouts/16v_arch.pdf).
- Microsoft (n.d.). *SIMD (Single instruction, multiple data)*. Accessed: 2017-07-05. URL: <http://microsoft.github.io/dotnet-features/features.html#8-9>.
- Intel (n.d.[a]). *Competitive Performance Summary with Intel® Xeon Phi™ Product Family*. Accessed: 2017-07-09. URL: <https://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-competitive-performance.html>.
- Rodinia (n.d.). *Rodinia : Accelerating Compute-Intensive Applications with Accelerators*. Accessed: 2017-07-10. URL: [https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating\\_Compute-Intensive\\_Applications\\_with\\_Accelerators](https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators).
- OpenDwarfs (n.d.[a]). *OpenDwarfs*. Accessed: 2017-07-10. URL: <https://github.com/vtsynergy/OpenDwarfs>.
- GPUOpen (n.d.). *CodeXL*. URL: <http://gpuopen.com/compute-product/codexl/>.
- Intel (2007). *Intel C++ Intrinsic Reference*. URL: <https://software.intel.com/sites/default/files/a6/22/18072-347603.pdf>.
- ICL (n.d.). *PAPI*. URL: <http://icl.cs.utk.edu/papi/index.html>.
- Bienia, Christian (2009). *Manpage of PARSEC MGMT*. URL: <http://parsec.cs.princeton.edu/doc/man/man1/parsecgmt.1.html>.

- UniProt (n.d.). *UniProt Knowledgebase*. URL: <http://www.uniprot.org/uniprot/>.
- Koby, Tim (n.d.). *N-body Simulations*. Accessed: 2017-07-24. URL: <http://physics.princeton.edu/~fpretori/Nbody/intro.htm>.
- Davies, Josh (n.d.). *2d n-body solver, with OpenMP, MPI, AVX*. Accessed: 2017-07-24. URL: <https://github.com/jodavies/nbody>.
- Shimpi, Anand Lal (2012). *Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel*. Accessed: 2017-07-20. URL: <http://www.anandtech.com/show/6355/intels-haswell-architecture>.
- Tomás, Pedro (2014). Accessed: 2017-07-22. URL: <https://fenix.tecnico.ulisboa.pt/downloadFile/1689468335556986/Lesson%2013%20-%20Modern%20Intel%20Processors.pdf>.
- Intel (n.d.[b]). *Intel Intrinsic Guide*. Accessed: 2017-07-22. URL: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.
- Q27 (n.d.). *27-Queens Puzzle: Massively Parellel Enumeration and Solution Counting*. Accessed: 2017-06-21. URL: <https://github.com/preusser/q27>.
- Al-Karadaghi, Salam (n.d.). *Sequence Alignment and Analysis*. Accessed: 2017-06-19. URL: <http://www.proteinstructures.com/Sequence/Sequence/sequence-alignment.html>.
- Roberts, Eric (n.d.). *Smith-Waterman Algorithm*. Accessed: 2017-06-19. URL: [https://cs.stanford.edu/people/eroberts/courses/soco/projects/computers-and-the-hgp/smith\\_waterman.html](https://cs.stanford.edu/people/eroberts/courses/soco/projects/computers-and-the-hgp/smith_waterman.html).
- Šošić, Martin (n.d.). *Opal: SIMD C/C++ library for massive optimal sequence alignment*. Accessed: 2017-07-24. URL: <https://github.com/Martinos/opal>.
- OpenDwarfs (n.d.[b]). *Cyclic Redundancy Check*. Accessed: 2017-07-23. URL: <https://github.com/vtsynergy/OpenDwarfs/tree/master/combinational-logic/crc>.
- Suresh, Anand (n.d.). *SSE4-CRC32*. Accessed: 2017-07-23. URL: [https://github.com/anandsuresh/sse4\\_crc32](https://github.com/anandsuresh/sse4_crc32).
- OpenDwarfs (n.d.[c]). *Baum-Welch Algorithm*. Accessed: 2017-07-23. URL: <https://github.com/vtsynergy/OpenDwarfs/tree/master/graphical-models/hmm>.
- Sand, Andreas (n.d.). *HMMlib*. Accessed: 2017-07-23. URL: <https://github.com/muldvang/thesis-code>.
- Huo, Yuchen and Guo, Danhao (n.d.). *parahmm: Parallel Implementation of HMM on Multicore Platform*. Accessed: 2017-07-23. URL: <https://github.com/firebb/parahmm>.
- Cisco (n.d.). *OpenH264*. Accessed: 2017-07-23. URL: <https://github.com/cisco/openh264>.
- Ghosh, Moinak (n.d.). *Pcompress*. Accessed: 2017-07-23. URL: <https://github.com/moinakg/pcompress>.
- Wikipedia (2017). *AVX-512*. Accessed: 2017-07-23. URL: <https://en.wikipedia.org/wiki/AVX-512>.
- Fog, Agner (2013). *Future instruction set: AVX-512*. Accessed: 2017-07-23. URL: <http://www.agner.org/optimize/blog/read.php?i=288>.
- Stephens, Nigel (2016). *Technology Update: The Scalable Vector Extension (SVE) for the ARMv8-A architecture*. Accessed: 2017-07-23. URL: <https://community.arm.com/processors/b/blog/posts/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture>.
- McCutchan, John (2013). *Using SIMD in Dart*. Accessed: 2017-07-17. URL: <https://www.dartlang.org/articles/dart-vm/simd>.

Lund, Kasper and Bak, Lars (2013). *Web Languages and VMs: Fast Code is Always in Fashion. (V8, Dart) - Google I/O 2013*. Video presentation. URL: <https://youtu.be/huawCR1o9H4>.

## Image Sources

CPU/GPU architecture comparison (n.d.). URL: [http://blog.goldenhelix.com/wp-content/uploads/2010/10/cpu\\_vs\\_gpu.png](http://blog.goldenhelix.com/wp-content/uploads/2010/10/cpu_vs_gpu.png).

N-body simulation (n.d.). URL: [https://upload.wikimedia.org/wikipedia/commons/3/37/Barnes-Hut\\_N-body\\_simulation\\_without\\_overlay.png](https://upload.wikimedia.org/wikipedia/commons/3/37/Barnes-Hut_N-body_simulation_without_overlay.png).

Chessboard drawing (n.d.). URL: <http://3.bp.blogspot.com/-LHJpIxH7BCY/TWEJc6koEhI/AAAAAAAAABM/2NIirDzATTI/w1200-h630-p-k-no-nu/8+vezir.png>.

Sequence alignment illustration (n.d.). URL: <http://www.genomecompiler.com/wp-content/uploads/2015/12/Sequence-Alignment.png>.

# Appendices





# Appendix A: Figures for Small and Large Inputs

Throughout the thesis, evaluation data has been presented for the medium input size. This appendix presents figures from the small and large input sizes for comparison.

## 1.1 N-body

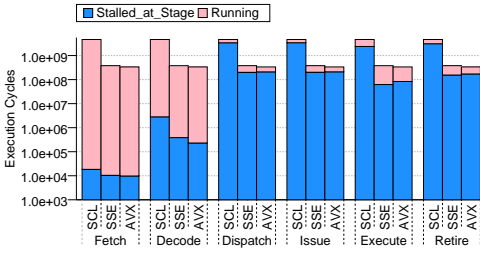


FIGURE A.1: N-BODY EXECUTION CYCLES (SMALL INPUT, LOG SCALE)

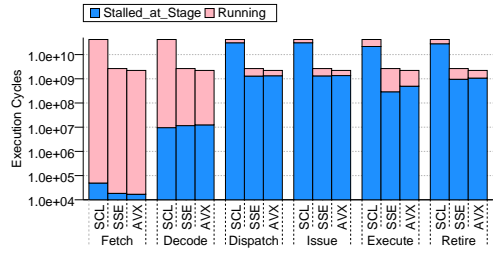


FIGURE A.2: N-BODY EXECUTION CYCLES (LARGE INPUT, LOG SCALE)

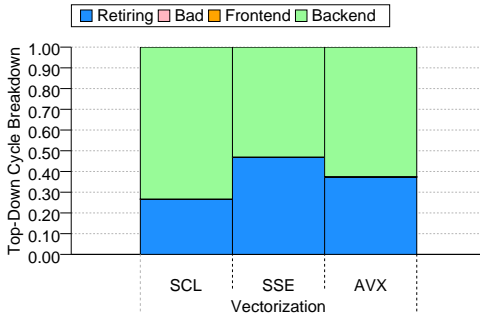


FIGURE A.3: N-BODY TOP-DOWN CYCLE BREAKDOWN (SMALL INPUT)

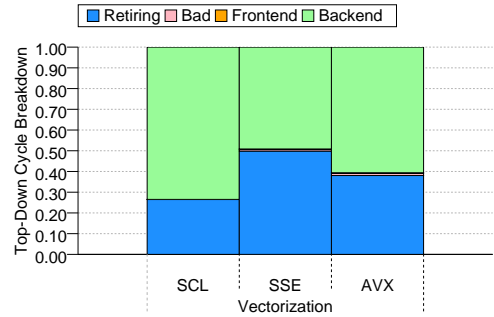


FIGURE A.4: N-BODY TOP-DOWN CYCLE BREAKDOWN (LARGE INPUT)

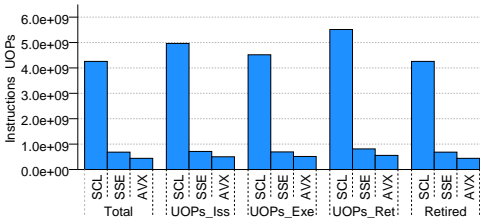


FIGURE A.5: N-BODY INSTRUCTIONS PER STAGE (SMALL INPUT)

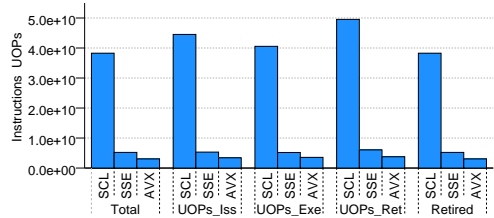


FIGURE A.6: N-BODY INSTRUCTIONS PER STAGE (LARGE INPUT)

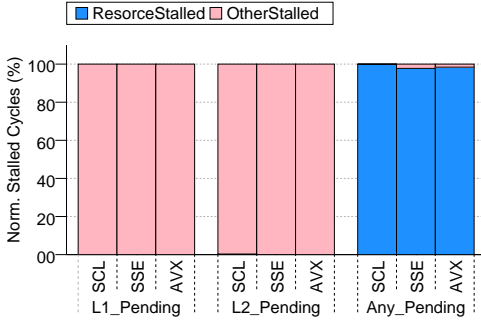


FIGURE A.7: N-BODY NORMALIZED STALLED CYCLES (SMALL INPUT)

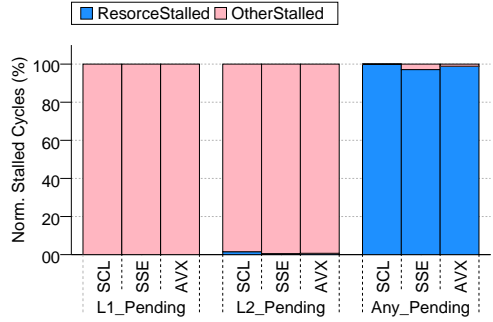


FIGURE A.8: N-BODY NORMALIZED STALLED CYCLES (LARGE INPUT)

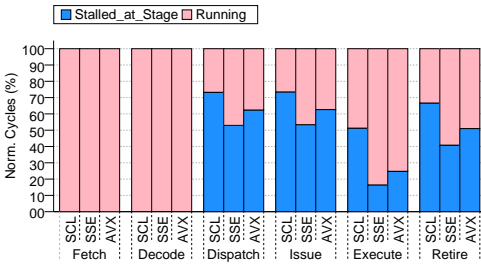


FIGURE A.9: N-BODY NORMALIZED STALLS PER STAGE (SMALL INPUT)

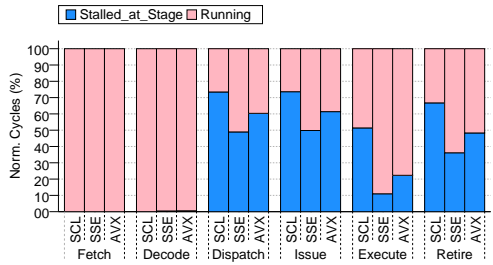


FIGURE A.10: N-BODY NORMALIZED STALLS PER STAGE (LARGE INPUT)

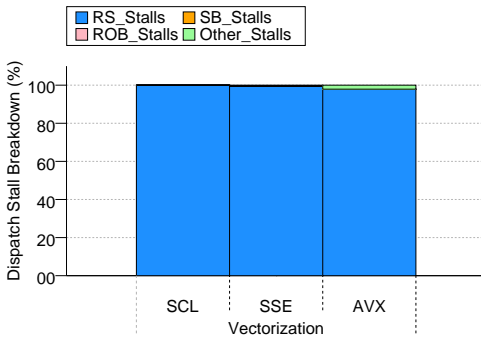


FIGURE A.11: N-BODY DISPATCH STALLS BREAKDOWN (SMALL INPUT)

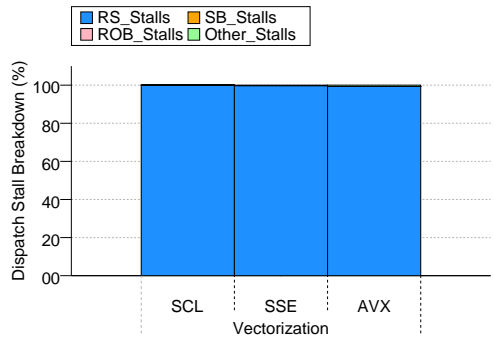


FIGURE A.12: N-BODY DISPATCH STALLS BREAKDOWN (LARGE INPUT)

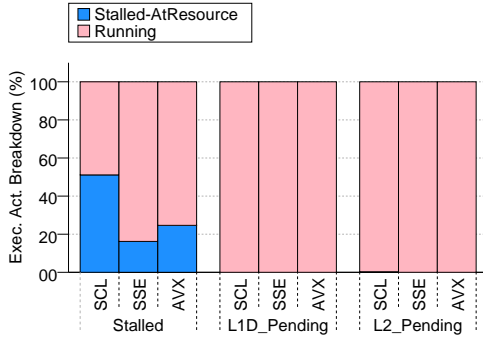


FIGURE A.13: N-BODY EXECUTION ACTIVITY BREAKDOWN (SMALL INPUT)

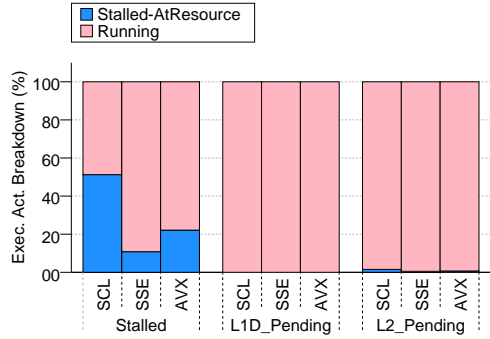


FIGURE A.14: N-BODY EXECUTION ACTIVITY BREAKDOWN (LARGE INPUT)

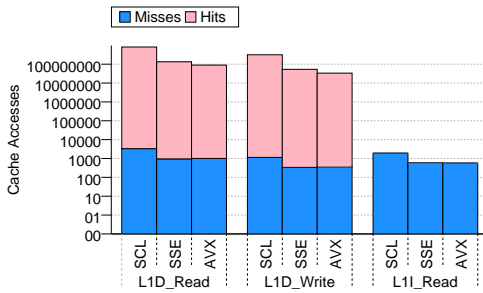


FIGURE A.15: N-BODY L1 ACCESSES (SMALL INPUT, LOG SCALE)

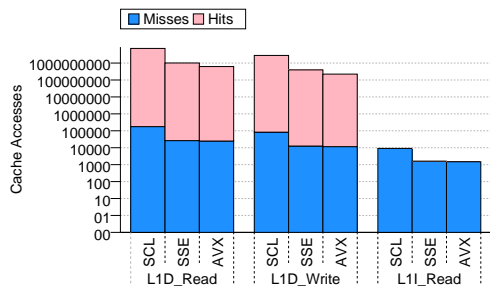


FIGURE A.16: N-BODY L1 ACCESSES (LARGE INPUT, LOG SCALE)

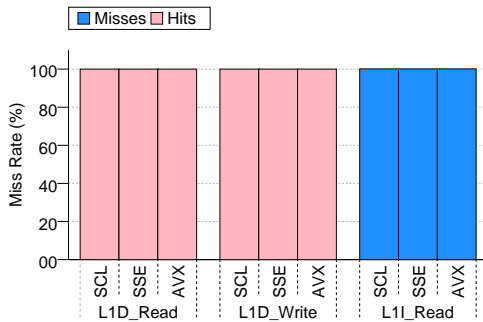


FIGURE A.17: N-BODY L1 MISS RATE (SMALL INPUT)

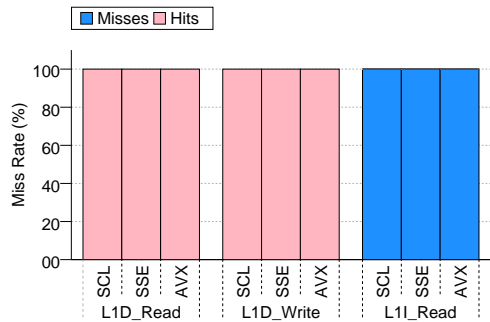


FIGURE A.18: N-BODY L1 MISS RATE (LARGE INPUT)

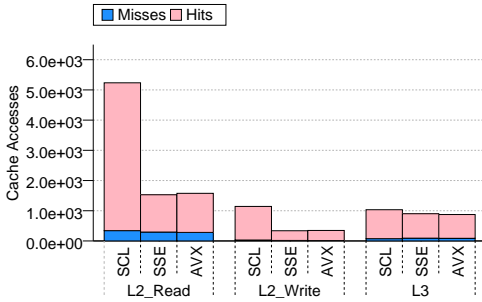


FIGURE A.19: N-BODY L2 AND L3 ACCESSES (SMALL INPUT, LOG SCALE)

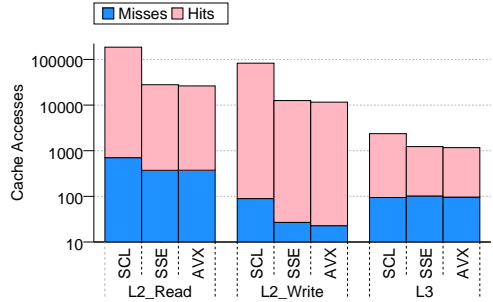


FIGURE A.20: N-BODY L2 AND L3 ACCESSES (LARGE INPUT, LOG SCALE)

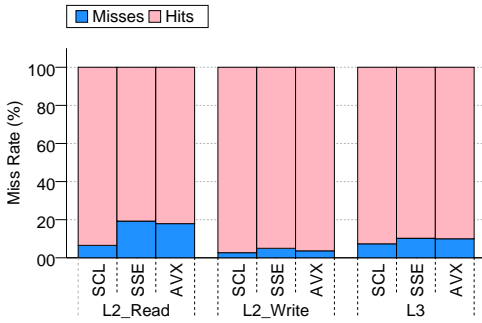


FIGURE A.21: N-BODY L2 AND L3 MISS RATES (SMALL INPUT)

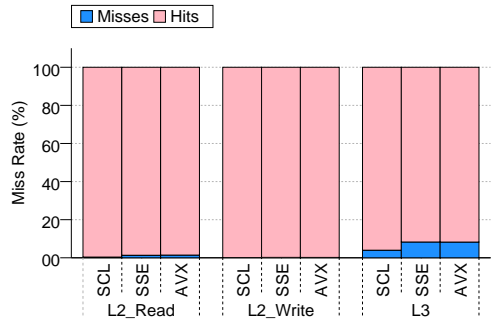


FIGURE A.22: N-BODY L2 AND L3 MISS RATES (LARGE INPUT)

## 1.2 N-Queens

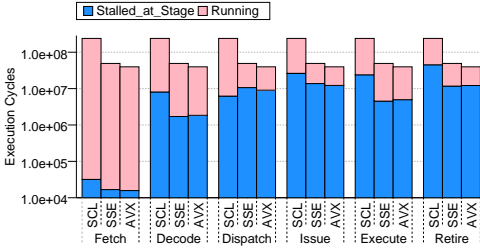


FIGURE A.23: N-QUEENS EXECUTION CYCLES (SMALL INPUT, LOG SCALE)

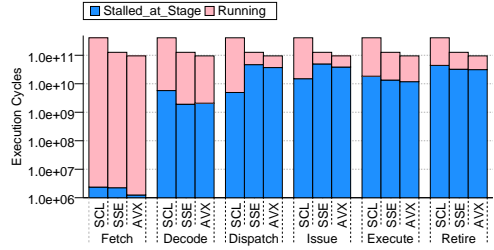


FIGURE A.24: N-QUEENS EXECUTION CYCLES (LARGE INPUT, LOG SCALE)

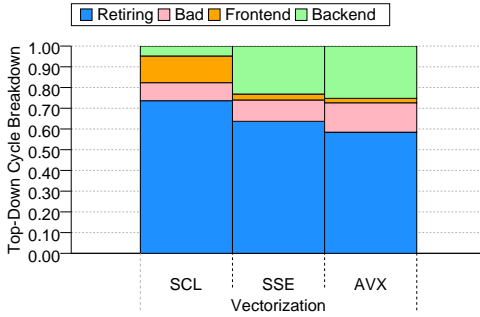


FIGURE A.25: N-QUEENS TOP-DOWN CYCLE BREAKDOWN (SMALL INPUT)

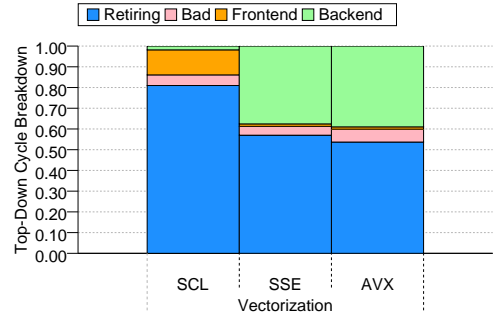


FIGURE A.26: N-QUEENS TOP-DOWN CYCLE BREAKDOWN (LARGE INPUT)

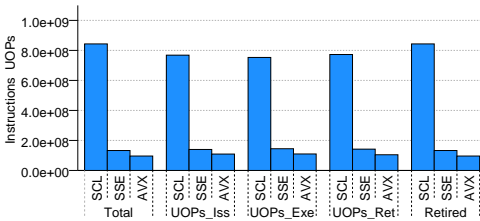


FIGURE A.27: N-QUEENS INSTRUCTIONS PER STAGE (SMALL INPUT)

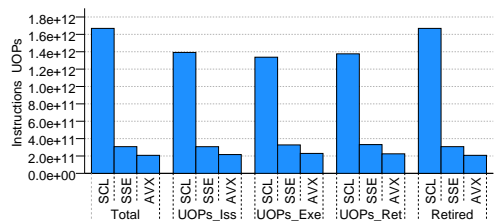


FIGURE A.28: N-QUEENS INSTRUCTIONS PER STAGE (LARGE INPUT)

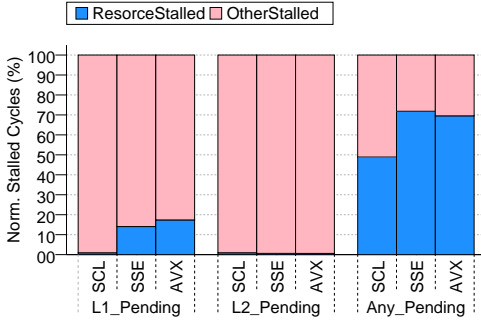


FIGURE A.29: N-QUEENS NORMALIZED STALLED CYCLES (SMALL INPUT)

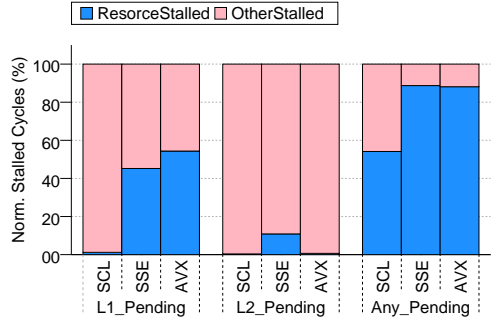


FIGURE A.30: N-QUEENS NORMALIZED STALLED CYCLES (LARGE INPUT)

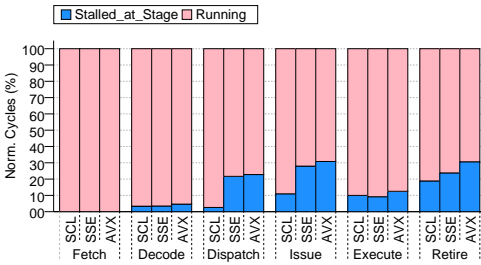


FIGURE A.31: N-QUEENS NORMALIZED STALLS PER STAGE (SMALL INPUT)

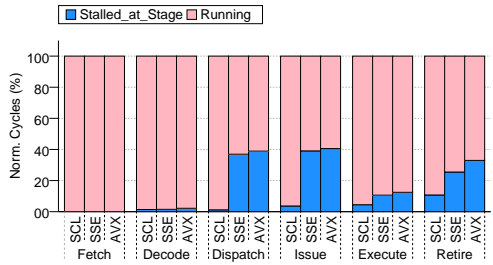


FIGURE A.32: N-QUEENS NORMALIZED STALLS PER STAGE (LARGE INPUT)

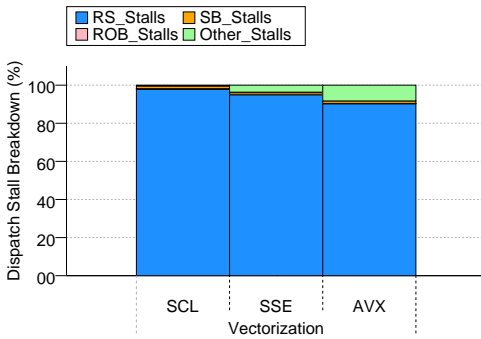


FIGURE A.33: N-QUEENS DISPATCH STALLS BREAKDOWN (SMALL INPUT)

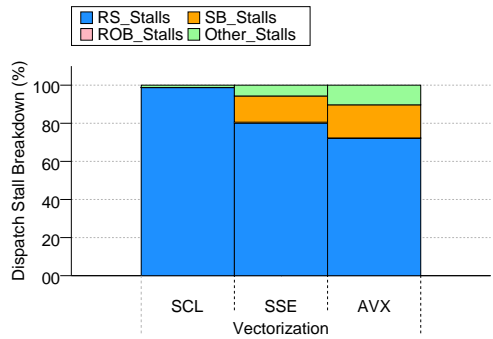


FIGURE A.34: N-QUEENS DISPATCH STALLS BREAKDOWN (LARGE INPUT)

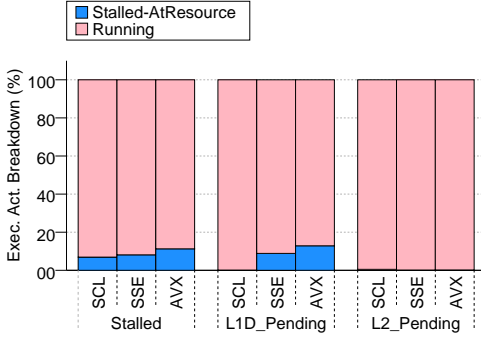


FIGURE A.35: N-QUEENS EXECUTION ACTIVITY BREAKDOWN (SMALL INPUT)

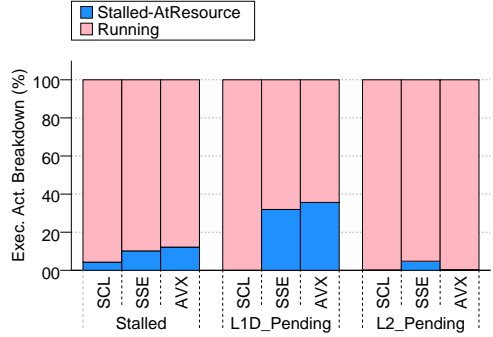


FIGURE A.36: N-QUEENS EXECUTION ACTIVITY BREAKDOWN (LARGE INPUT)

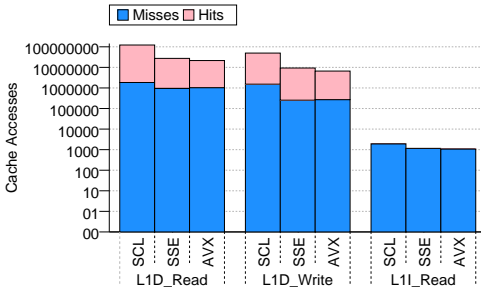


FIGURE A.37: N-QUEENS L1 ACCESSES (SMALL INPUT, LOG SCALE)

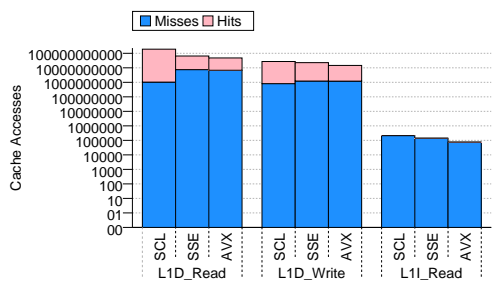


FIGURE A.38: N-QUEENS L1 ACCESSES (LARGE INPUT, LOG SCALE)

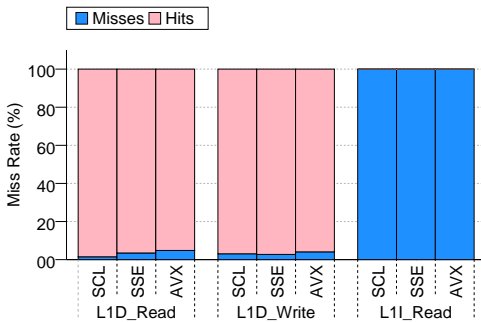


FIGURE A.39: N-QUEENS L1 MISS RATE (SMALL INPUT)

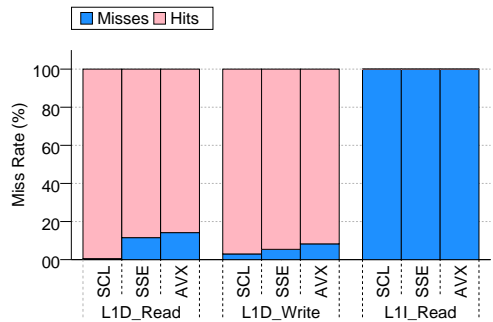


FIGURE A.40: N-QUEENS L1 MISS RATE (LARGE INPUT)





FIGURE A.41: N-QUEENS L2 AND L3 ACCESSES (SMALL INPUT, LOG SCALE)

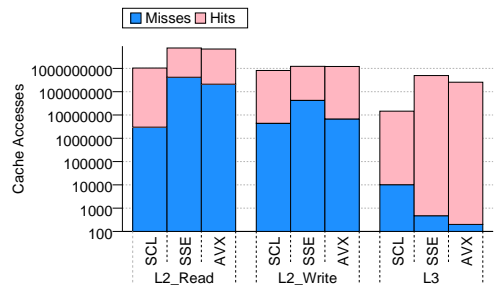


FIGURE A.42: N-QUEENS L2 AND L3 ACCESSES (LARGE INPUT, LOG SCALE)

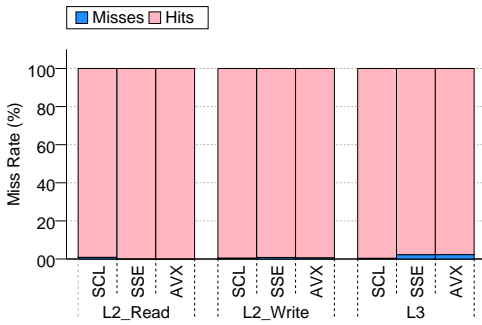


FIGURE A.43: N-QUEENS L2 AND L3 MISS RATES (SMALL INPUT)

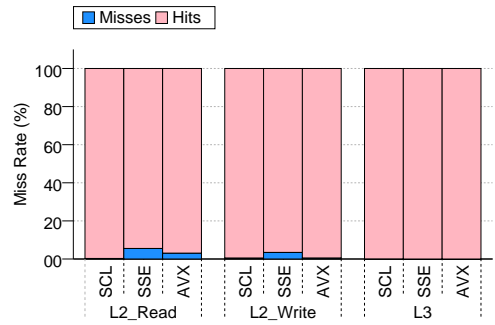


FIGURE A.44: N-QUEENS L2 AND L3 MISS RATES (LARGE INPUT)

### 1.3 Needleman-Wunsch

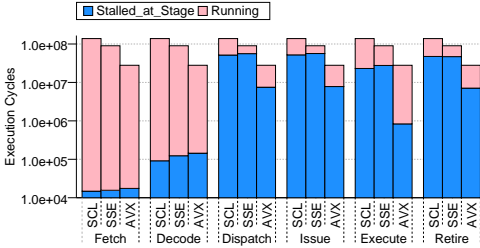


FIGURE A.45: NW EXECUTION CYCLES (SMALL INPUT, LOG SCALE)

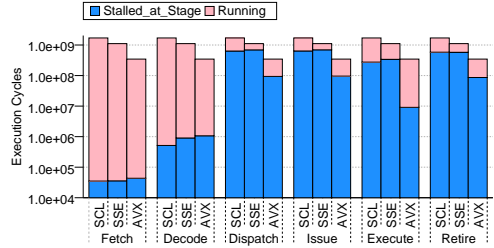


FIGURE A.46: NW EXECUTION CYCLES (LARGE INPUT, LOG SCALE)

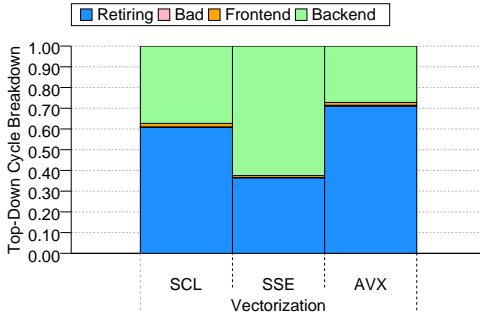


FIGURE A.47: NW TOP-DOWN CYCLE BREAKDOWN (SMALL INPUT)

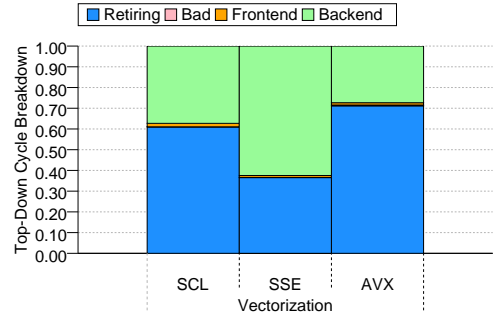


FIGURE A.48: NW TOP-DOWN CYCLE BREAKDOWN (LARGE INPUT)

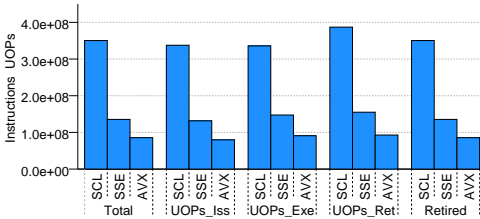


FIGURE A.49: NW INSTRUCTIONS PER STAGE (SMALL INPUT)

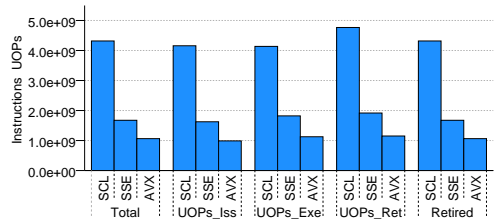


FIGURE A.50: NW INSTRUCTIONS PER STAGE (LARGE INPUT)

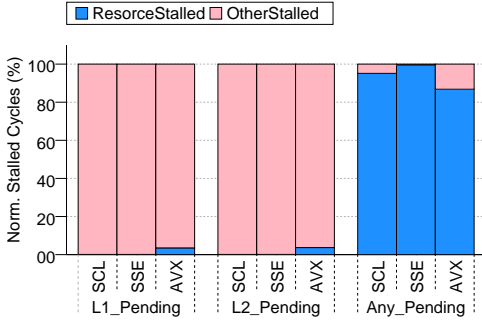


FIGURE A.51: NW NORMALIZED STALLED CYCLES (SMALL INPUT)

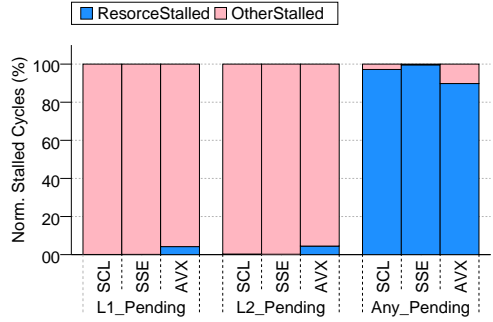


FIGURE A.52: NW NORMALIZED STALLED CYCLES (LARGE INPUT)

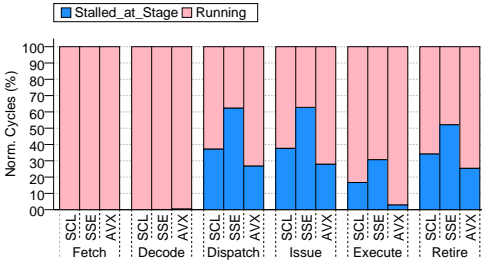


FIGURE A.53: NW NORMALIZED STALLS PER STAGE (SMALL INPUT)

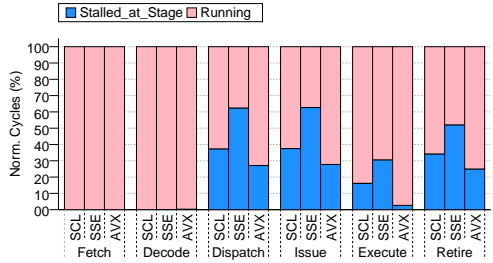


FIGURE A.54: NW NORMALIZED STALLS PER STAGE (LARGE INPUT)

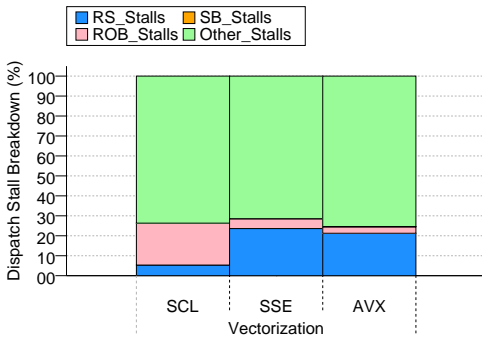


FIGURE A.55: NW DISPATCH STALLS BREAKDOWN (SMALL INPUT)

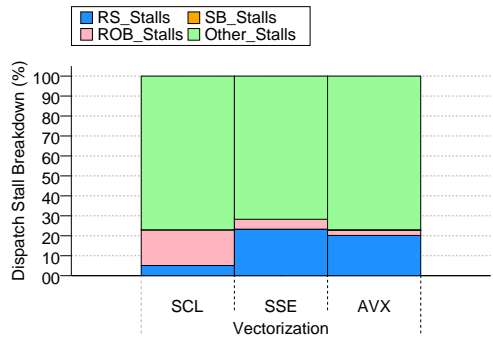


FIGURE A.56: NW DISPATCH STALLS BREAKDOWN (LARGE INPUT)

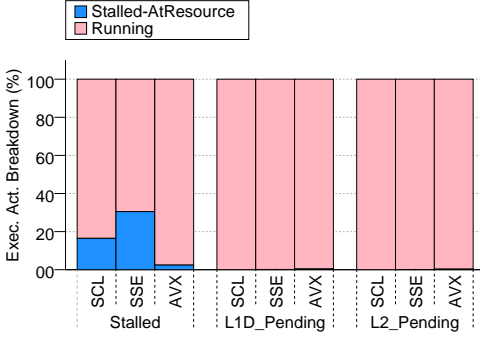


FIGURE A.57: NW EXECUTION ACTIVITY BREAKDOWN (SMALL INPUT)

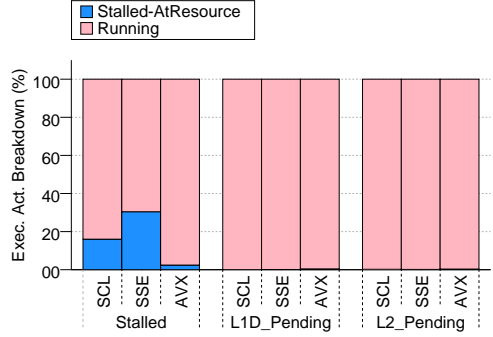


FIGURE A.58: NW EXECUTION ACTIVITY BREAKDOWN (LARGE INPUT)

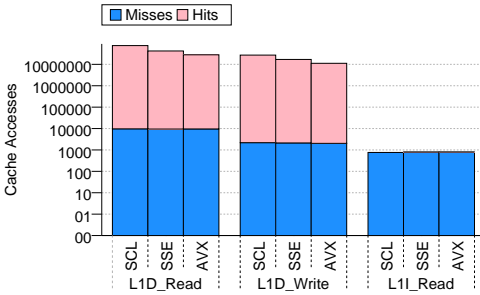


FIGURE A.59: NW L1 ACCESSES (SMALL INPUT, LOG SCALE)

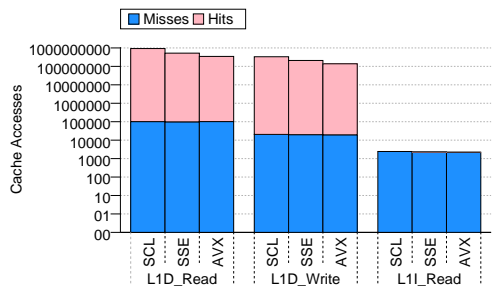


FIGURE A.60: NW L1 ACCESSES (LARGE INPUT, LOG SCALE)

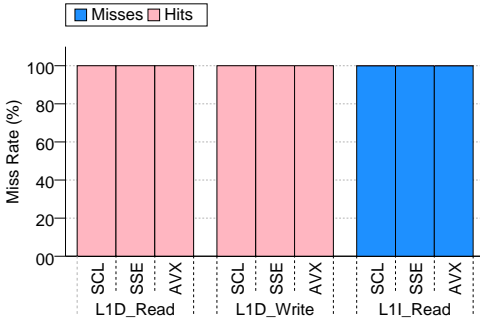


FIGURE A.61: NW L1 MISS RATE (SMALL INPUT)

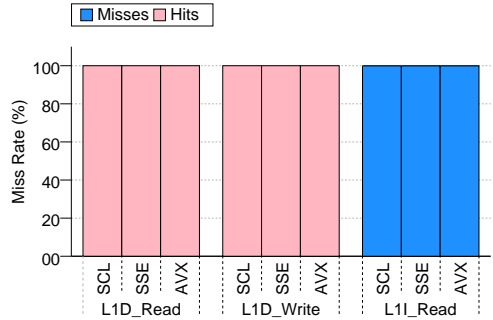


FIGURE A.62: NW L1 MISS RATE (LARGE INPUT)

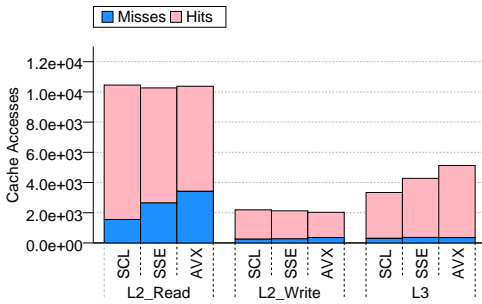


FIGURE A.63: NW L2 AND L3 ACCESSES (SMALL INPUT, LOG SCALE)

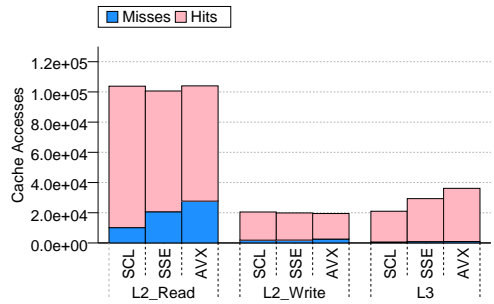


FIGURE A.64: NW L2 AND L3 ACCESSES (LARGE INPUT, LOG SCALE)

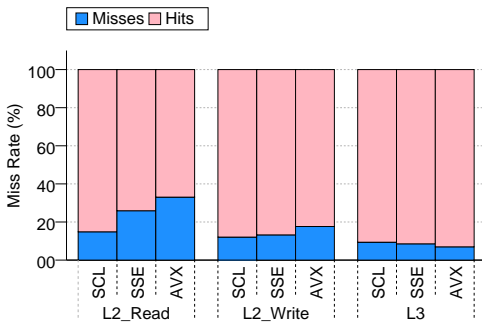


FIGURE A.65: NW L2 AND L3 MISS RATES (SMALL INPUT)

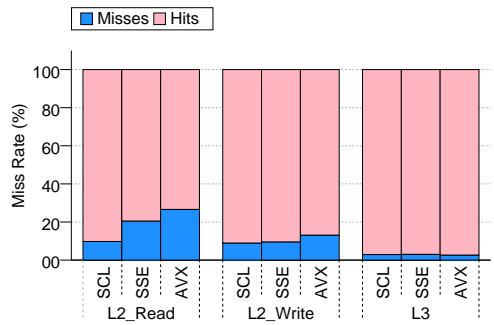


FIGURE A.66: NW L2 AND L3 MISS RATES (LARGE INPUT)

### 1.4 Smith-Waterman

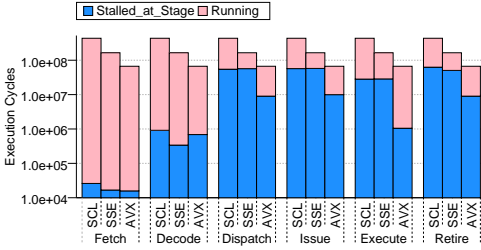


FIGURE A.67: SWAT EXECUTION CYCLES (SMALL INPUT, LOG SCALE)

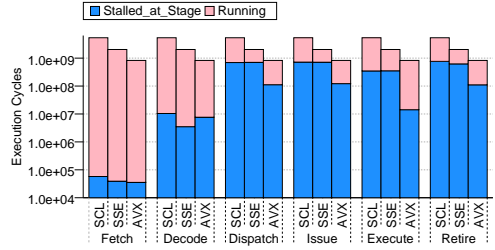


FIGURE A.68: SWAT EXECUTION CYCLES (LARGE INPUT, LOG SCALE)

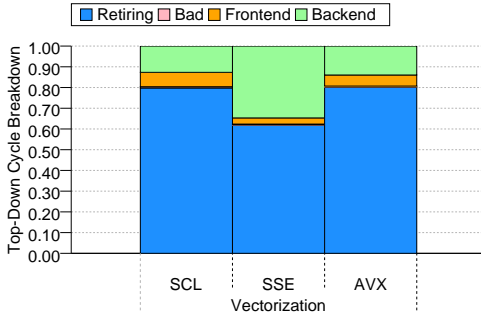


FIGURE A.69: SWAT TOP-DOWN CYCLE BREAKDOWN (SMALL INPUT)

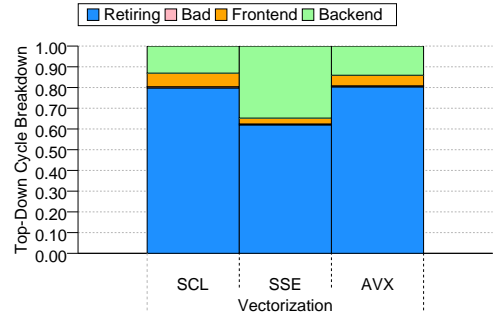


FIGURE A.70: SWAT TOP-DOWN CYCLE BREAKDOWN (LARGE INPUT)

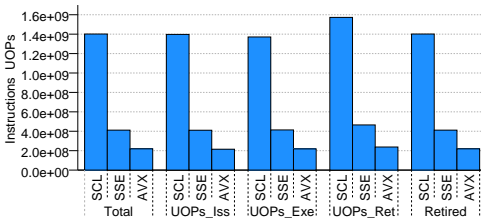


FIGURE A.71: SWAT INSTRUCTIONS PER STAGE (SMALL INPUT)

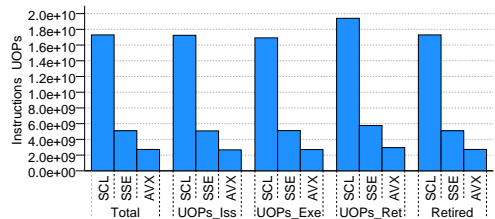


FIGURE A.72: SWAT INSTRUCTIONS PER STAGE (LARGE INPUT)

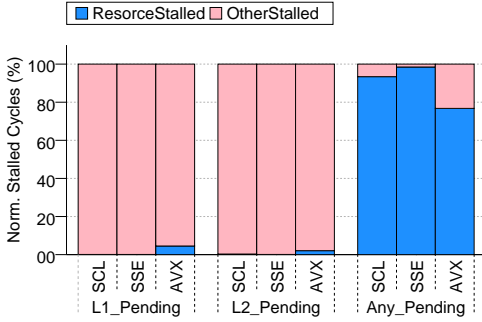


FIGURE A.73: SWAT NORMALIZED STALLED CYCLES (SMALL INPUT)

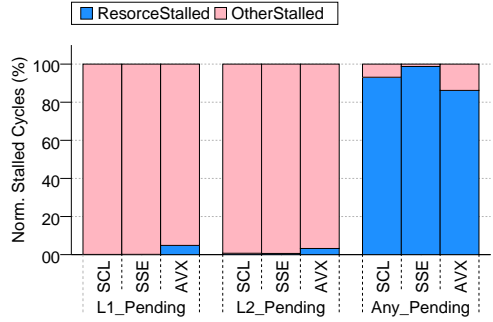


FIGURE A.74: SWAT NORMALIZED STALLED CYCLES (LARGE INPUT)

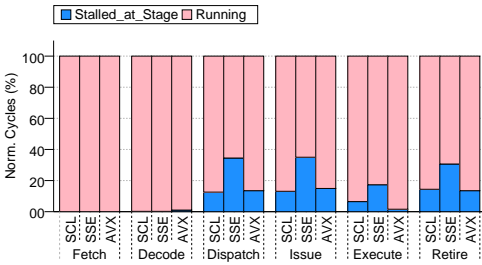


FIGURE A.75: SWAT NORMALIZED STALLS PER STAGE (SMALL INPUT)

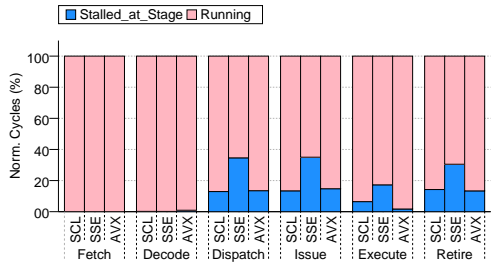


FIGURE A.76: SWAT NORMALIZED STALLS PER STAGE (LARGE INPUT)

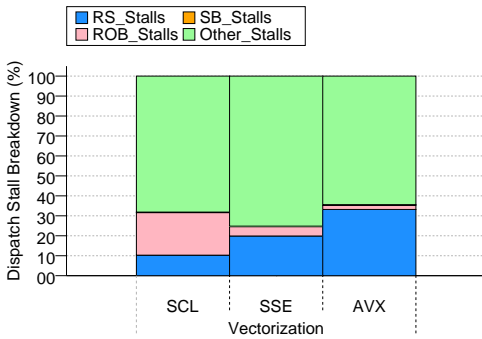


FIGURE A.77: SWAT DISPATCH STALLS BREAKDOWN (SMALL INPUT)

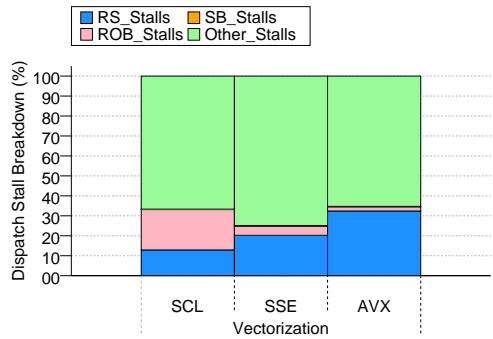


FIGURE A.78: SWAT DISPATCH STALLS BREAKDOWN (LARGE INPUT)

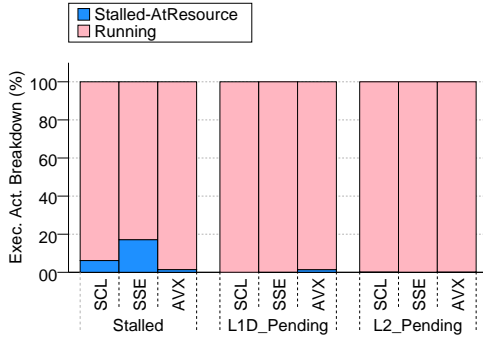


FIGURE A.79: SWAT EXECUTION ACTIVITY BREAKDOWN (SMALL INPUT)

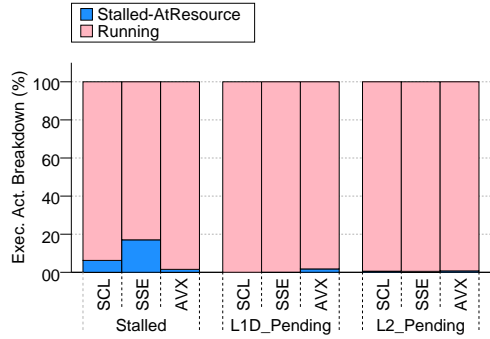


FIGURE A.80: SWAT EXECUTION ACTIVITY BREAKDOWN (LARGE INPUT)

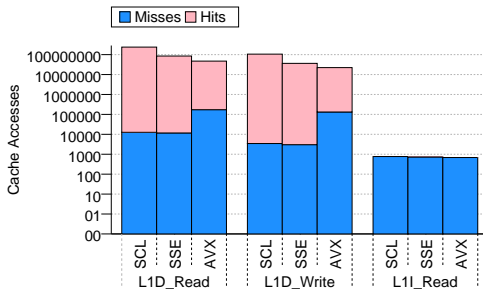


FIGURE A.81: SWAT L1 ACCESSES (SMALL INPUT, LOG SCALE)

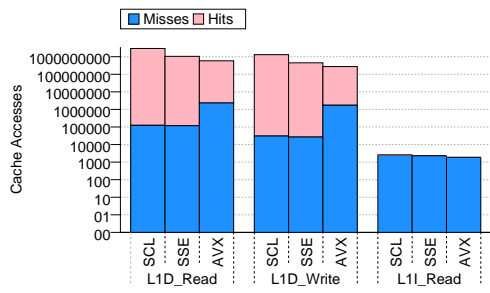


FIGURE A.82: SWAT L1 ACCESSES (LARGE INPUT, LOG SCALE)

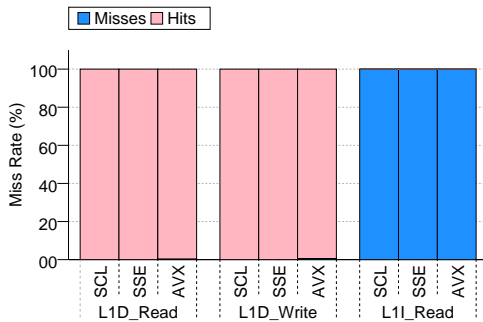


FIGURE A.83: SWAT L1 MISS RATE (SMALL INPUT)

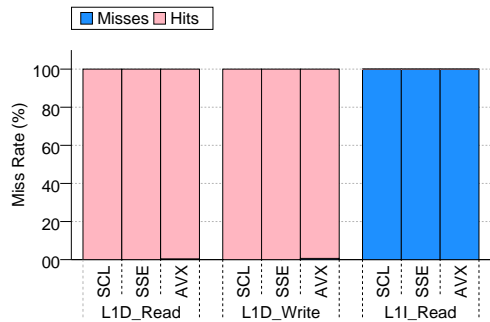


FIGURE A.84: SWAT L1 MISS RATE (LARGE INPUT)



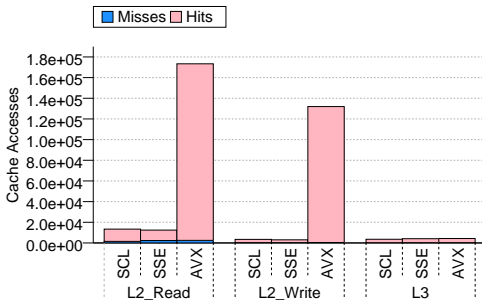


FIGURE A.85: SWAT L2 AND L3 ACCESSSES (SMALL INPUT, LOG SCALE)

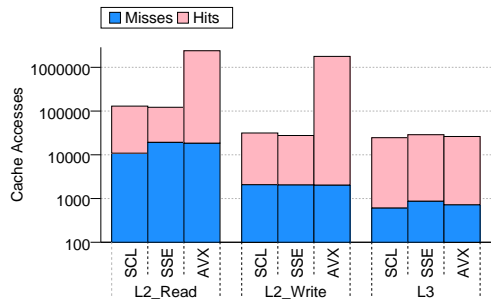


FIGURE A.86: SWAT L2 AND L3 ACCESSSES (LARGE INPUT, LOG SCALE)

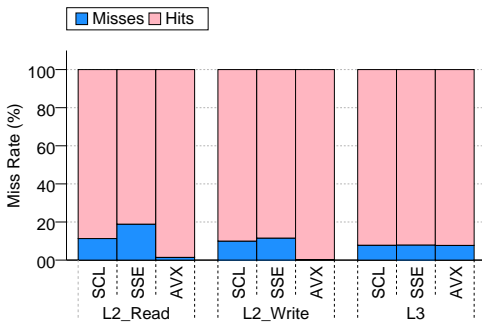


FIGURE A.87: SWAT L2 AND L3 MISS RATES (SMALL INPUT)

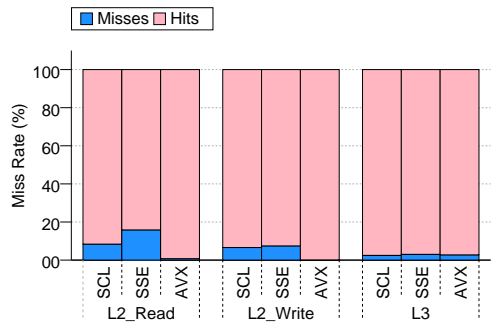


FIGURE A.88: SWAT L2 AND L3 MISS RATES (LARGE INPUT)