Problem Statement

Today, all the largest supercomputers are manufactured in the US and Asia, not in Europe. Mont Blanc is a newly started EU project aimed at putting Europe back on the map of supercomputer manufacturers. The goal is to build a prototype that will be ranked at the top of the Green500.org ranking list of the most energy efficient computers. Energy efficiency is already a main design constraint for supercomputers, and it is expected to become the dominating design challenge in the future. Mont Blanc will achieve energy efficiency by adopting low end processors from mobile phones and other embedded systems products. The project considers both GPUs from Nvidia, ARM-Nvidia Tegra processors and the Mali T604 (quad-core) and T658 (8-core) GPUs developed by ARM.

The main goal of the thesis is to do experiments and evaluate performance and energy-efficiency of applications using the OmpSs environment for parallel programming developed by UPC/BSC. OmpSs will be used in the Mont Blanc project and is also a central part of many other European EU-7FP research projects. Consequently, increased knowledge about OmpSs at NTNU makes it more likely that NTNU can contribute to Mont Blanc and other EU projects in the future. The following elements are included in the project:

a) Study the OmpSs environment for parallel programming.

b) Evaluate performance and energy-efficiency of OmpSs applications or application kernels. (The selection of applications is dependent of what OmpSs codes are made available to NTNU.)

c) Problems should be evaluated for different problem sizes at a desktop computer equipped with an Intel core i7 (Sandy Bridge) quad-core processor. Currently, energy measurements can be done by reading the Sandy Bridge MSR energy registers.

d) The effect of vectorization should be investigated.

e) If time permits, and if HW is made available by ARM, energy-efficiency studies of the same OmpSs codes on ARM HW should be conducted.

f) The report should document the work in a way that helps future students continue the with projects along the same path of research.

Acknowledgements

First of all I would like to thank professor Lasse Natvig at NTNU for being my primary supervisor for my thesis, and giving me excellent supervision. He kept me motivated for working hard on this thesis by giving me very helpful feedback on my work, as well as providing interesting discussions and ideas at our meetings.

My thanks also goes to Jörg Wagner at the ARM office in Trondheim. He not only gave us access to ARM hardware by arranging a loan agreement between NTNU and ARM, but he also spent a significant amount of time to set up the hardware so that it was ready to use. Additionally, he gave valuable feedback on the final results.

Additionally, I would like to direct my sincere gratitudes to Jan Christian Meyer. Jan Christian spent a significant amount of time reading through my final report, and giving very high quality and useful feedback. Thanks to him, the quality of the report was significantly improved.

Abstract

In this thesis, the performance and energy efficiency of two current hardware platforms are evaluated, the Intel Sandy Bridge Core i7 and ARM Cortex-A9 MPCore test chip, using techniques like vectorization and multi-threading with task-based programming using OmpSs. We present results from three task-based programs, Black-Scholes, FFTW and matrix multiplication on both platforms. The performance and energy efficiency is compared between different configurations of threads, vectorization and task scheduling algorithms. Energy consumption is measured using the Running Average Power Limit interface on the Sandy Bridge, and regular sampling of the current power dissipation from the system configuration registers on ARM. The energy efficiency results are presented using the metrics total energy consumed, power, GFLOPS/W, and the energy-delay and energy-delay squared products. The energy efficiency of the ARM Cortex-A9 MPCore is compared to that of Sandy Bridge using the process-normalized energy-delay and energy-delay squared products, as well as GFLOPS/W and energy.

Black-Scholes is adapted to use vector code, and FFTW is compiled with and without vector support. The matrix multiplication application uses ATLAS, which already is vectorized. Code for sampling and numerically integrating the power dissipation over time was developed for ARM, then different task scheduling algorithms are explored for each application.

For both platforms, vectorization with SSE/AVX and NEON is found to consume little to no extra energy per second while giving significantly higher performance. Multi-threading gives higher performance, but with higher power consumption. With AVX on Intel, Black-Scholes shows an energy efficiency of 0.82 GFLOPS/W, FFTW up to 1.4 GFLOPS/W, and matrix multiplication almost 2 GFLOPS/W. Both multi-threading and vectorization significantly reduced the energy-delay products, showing up to 99.55% reduction for Black-Scholes and 93.65% for FFTW compared to non-vectorized single-threaded code for the Intel platform. The ARM results are restricted from publishing, but can be found in appendix A for those with access.

Single-threaded execution is shown to give a better GFLOPS/W for small problem sizes in the benchmarked applications, while the EDP is reduced for multi-threading even for small problems. Black-Scholes show increased performance and energy efficiency with hyper-threading. FFTW shows no significant difference using hyper-threads compared to four threads. The performance and energy efficiency for matrix multiplication is lower when hyper-threading is used.

Abstract (Norwegian)

I denne oppgaven evalueres ytelse og energieffektivitet for to moderne maskinvareplatformer, Intel Sandy Bridge Core i7 og ARM Cortex-A9 MPCore test chip. Teknikker som vektorisering og multitråding brukes sammen med task-basert programmering i OmpSs. Vi presenterer resultater fra tre task-baserte programmer, Black-Scholes, FFTW og matrisemultiplikasjon på begge plattformer hvor ytelse og energieffektivitet sammenlignes for forskjellige konfigurasjoner av tråder, vektorisering og task-scheduling-algoritmer. Energibruk blir målt ved bruk av Running Average Power Limit-grensesnittet på Sandy Bridge, og ved jevnlig sampling av effekt ved bruk av systemkonfigurasjonsregistere på ARM. Resultater for energieffektivitet presenteres ved bruk av metrikker som total energibruk, effekt, GFLOPS/W, og energy-delay- og energy-delay-squared produkter. Energieffektiviteten på ARM Cortex-A9 MPCore sammenlignes med Sandy Bridge ved bruk av prosess-normalisert energy-delay og energy-delay squared, samt GFLOPS/W og energi.

Black-Scholes blir tilpasset til å bruke vektorkode, og FFTW ble kompilert med og uten vektorstøtte. Matrisemultiplikasjonsapplikasjonen bruker ATLAS som allerede er vektorisert. En algoritme for sampling og integrering av effekt ble utviklet for ARM. Forskjellige task-scheduling-algoritmer ble utforsket for å finne en optimal algoritme for hver applikasjon.

For begge plattformer viser vektorisering med SSE/AVX og NEON seg å bruke fra liten til ingen ekstra energi per tidsenhet sammenlignet med ikke-vektorisert kode. Multitråding gir høyere ytelse, men også et betydelig høyere effektforbruk. Med AVX på Intel-plattformen viser Black-Scholes en energieffektivitet på 0.82 GFLOPS/W, FFTW up til 1.4 GFLOPS/W, og matrisemultiplikasjon nesten 2 GFLOPS/W. Både multithreading og vektorisering reduserte energy-delay-produktet betydelig, med opp til 99.55% reduksjon for Black-Scholes og 93.65% for FFTW sammenlignet med ikke-vektorisert entrådet kode på Intel-plattformen. ARM-resultatene er unntatt publisering, men finnes i appendix A for de med tilgang.

Kjøring på èn tråd viser seg å gi høyere GFLOPS/W for små problemstørrelser i applikasjonene som er testet, mens EDP er redusert med flertråding selv for små problemer. Black-Scholes viser økt ytelse og energieffektivitet med hyper-tråding. FFTW viser ingen vesentlig forskjell ved å bruke hypertråder. Både ytelse og energieffektivitet for matrisemultiplikasjon er lavere når hyper-tråding er brukt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, the motivation and goals for the thesis is presented, then the contributions from this work are outlined. Additionally, some terminology will be explained, and an outline of the chapters is presented.

1.1 Motivation

Energy requirements is one of the major challenges for large supercomputers [2]. If the most powerful supercomputer as per November 2011 were scaled to one exaFLOPS without improving energy efficiency, it would require more than a gigawatt of power, enough to power almost a million homes. Not only is this impractical due to the cost of operation, but each component must also be cooled which also incurs a significant cost in equipment and additional energy use, further increasing operational costs.

The Mont Blanc project is an EU project aimed at reaching the exaFLOPS milestone by using energy efficient embedded accelerators, for instance the ARM Mali GPUs, in order to get the energy efficiency of the computer up to a level where this is possible. The first goal is 7 GFLOPS/W and a total performance of 50 petaFLOPS by 2014. For 2017, the goal is 20 GFLOPS/W by 2017, approaching the goal of 50 GFLOPS/W required for an exaFLOPS computer with a power budget of 20MW.

Although much of the improvement must come from advances in hardware, the software also matters. Often, energy efficiency and performance are closely related, in that higher performance often gives higher energy efficiency, given that the hardware configuration stays the same. However, this may not always be true; in this project it was found that vectorizing codes consumes little to no extra energy for a significant performance boost, while multi-threading involves activating more cores, incurring significantly higher energy usage.

This thesis is the result of the first in a series of research projects exploring energy efficiency in modern computer architectures. First we present the results from strictly on-chip energy usage, where most of the execution occurs on-chip with minimal interaction from the memory system. Future research projects will measure the energy consumption of the full computing system, before expanding to energy measurements on multi-node computing clusters. The ultimate goal is to try to understand the energy implications of every subsystem, and use that knowledge to develop methods for more energy efficient computers, both through improvements in software and in hardware.

1.2 Thesis Scope and Goals

The goal of this thesis is to evaluate current hardware platforms with respect to energy efficiency and performance using techniques like vectorization and multi-threading with task based programming using OmpSs. The CPUs used in the experiments are the Intel Sandy Bridge CPU Core i7 2600 and a development implementation of the ARM Cortex-A9 MPCore quad-core CPU. The Sandy Bridge architecture is the latest generation of Intel CPUs before Ivy Bridge, which was released in April 2012, and the Cortex-A9 is relevant in many embedded devices like cell phones. Before evaluating energy efficiency, the applications are first optimized for performance.

It is also of interest to compare the ARM and Intel platforms, in particular because the Sandy Bridge is a relatively new architecture that is claimed to reach "new levels of performance, flexibility and energy efficiency" by Intel [3], and ARM is relevant in many mobile and embedded devices where energy efficiency is a key point.

1.3 Terminology

1.3.1 Performance Measurements and FLOPS

In this work, FLOPS with capital letters is used to refer to the rate of computation, i.e. FLoating-point OPerations per Second. When talking about the quantity "number of floating point operations", either "floating point operation(s)", or "flop(s)" in lower case, is be used.

1.4 Contributions

The contributions of this work are:

1. Performance and energy efficiency results for three applications, Black-Scholes, FFTW and matrix multiplication for both ARM Cortex-A9 MPCore test chip and Intel Core i7 2600.

2. Comparisons of the energy efficiency for these two CPUs using the metrics energy, GFLOPS/W, process-normalized energy-delay and energy-delay squared product.

3. Models for performance in task-based programs based on task scheduling overhead and memory overhead.

4. A research paper presenting the Sandy Bridge results with the GFLOPS/W metric. The paper is going to be published in the LNCS 7453 proceedings, and presented at the EECS section of the ICT-GLOW 2012 conference. It is attached in appendix H.

5. A benchmarking framework that was developed because of the large number of benchmarks that was run and had to be managed throughout the writing of this thesis. This framework is briefly described in appendix G.

The division of work for the research paper is as follows: Hallgeir Lien provided the results, the research infrastructure and wrote several parts of the paper, especially the discussion. Lasse Natvig provided the main paper structure, and wrote other parts of the paper, using the then current version of Hallgeir Lien's thesis as main source. Abdullah Al Hasib started the work on energy measurements using MSRs and did some contributions to the text, while Jan Christian Meyer acted both as scientific discussion partner in the final stages of the work, and also helped improving the text before submission and for the camera ready copy. The work will be presented at ICT-GLOW in September 2012.

1.5 Thesis Outline

This thesis is structured as follows. Chapter 2 gives a background in the central topics of this paper: Mont Blanc, OmpSs, challenges in large supercomputers, heterogeneous computing, vectorization (SSE, AVX and NEON), energy metrics, and related work. Chapter 3 give an overview of the benchmarked applications, Black-Scholes, FFTW and matrix multiplication. Chapter 4 explains the implementation details for each application or benchmark. Chapter 5 covers methodology and experiment setup: Compiler flags used, software versions, hardware, statistical metrics and how energy measurements were performed. Chapter 6 presents the performance and energy efficiency results for all three applications, with some discussion of each result. Chapter 7 presents a derivation of a model for performance where task creation and scheduling overheads are taken into account, and a model estimating overhead of main memory accesses for each application. Chapter 8 concludes with a summary of the main results, and a conclusion. Appendix A presents the performance results for the ARM processor. The appendix is restricted from publishing, so it is provided as an attachment to this report.

Chapter 2

Background

In this chapter background on the tools and programming models used will be given. The Mont Blanc project is also presented. The chapter is organized as follows: Section 2.1 gives an overview of exascale computing, the Mont Blanc project and its goals. Section 2.2 gives an introduction to the OmpSs programming model developed at the Barcelona Supercomputing Center and the relevance it has for this thesis. Section 2.3 explains the concepts behind SIMD and how this is implemented in the Intel Sandy Bridge CPUs. Section 2.4 presents different metrics that are used for evaluating energy efficiency. Lastly, section 2.5 gives an overview of related work and their results.

2.1 The Mont Blanc Project and Exascale Computing

A major milestone for supercomputers is hitting the exaFLOPS mark in performance. Today's most powerful supercomputer is rated to just over 10 petaFLOPS [4]. A hundredfold improvement in performance is required to reach one exaFLOPS. This presents a number of challenges. The Mont Blanc project, a part of the European Exascale Software Initiative [5], is one of the projects trying to address some of the challenges in exascale computing.

2.1.1 Challenges for exascale supercomputers

Developing an exascale supercomputer requires significant technological advances in many areas [2]. For instance, the performance gap between memory, storage systems and CPUs is growing [6]. If memory systems and interconnects are unable to scale with the CPU development, bandwidth will be a serious bottleneck in an exascale computer.

Another challenge the supercomputing community is facing is the increased complexity of writing software for large computing clusters in the presence of the CPU-memory performance gap and heterogeneous systems. In order to minimize the performance impact of data movement, it has proved useful to overlap computation with communication time in clusters [7][8][9]. Some research has been conducted in automatically taking care of this overlap through alternative programming models. Tarragon

[10] developed at UC San Diego is one such approach that is an actor based execution model where the programmer defines a partial ordering of tasks, and the runtime system will then take care of scheduling these tasks on physical processors.

One of the most significant barriers that supercomputers face today is power dissipation [2]. The most powerful supercomputer as of November 2011 as listed by the Top 500 website [4], K Computer at the RIKEN Advanced Institute for Computational Sciences (AICS) in Japan, has a theoretical peak performance of 11.3 petaFLOPS and draws 12.66 megawatts of power, giving a energy efficiency of 0.89 GFLOPS/W. If we were to scale this linearly up to one exaFLOPS, we get a power requirement of 1.12 GW. On the other hand, using the most energy efficient supercomputer, the IBM BlueGene/Q, Power BQC at the IBM Rochester site [11] uses 2.02 GFLOPS/W, giving a power requirement of 495MW for one exaFLOPS. A more reasonable power budget is 20 MW [12], which would require an efficiency of 50 GFLOPS/W, a 25x increase in energy efficiency from the energy efficiency leader as per the Green500 list, November 2011 [11].

2.1.2 The Mont Blanc project

The Mont Blanc project, one of three projects under the European Exascale Software Initiative (EESI), is a project with the ultimate goal of building an exascale supercomputer. Mont Blanc has three main objectives: [1]

1. Deploying a prototype HPC system based on existing energy efficient embedded technologies, scalable to 50 petaFLOPS using only 7 MW of power, competitive on the Green 500 list in 2014.
2. Designing a next-generation HPC system based on new embedded technologies, overcoming some or most of the limitations of the previous generation. This system will be scalable to 200 petaFLOPS at 10 MW, competitive for the Top 500 list by 2017.
3. Porting and optimizing a small number of representative exascale applications for use on this new system, as well as some important scientific libraries. A preliminary list of these applications can be found in table 2.1, and the libraries have been listed in table 2.2.

As mentioned in the previous section, the most energy efficient supercomputer per November 2011 according to the Green 500 list has an efficiency of 2.02 GFLOPS/W [11]. However, the first objective of Mont Blanc is to achieve more than 7 GFLOPS/W, more than tripling this efficiency. The next objective by 2017 is improving this to 20 GFLOPS/W, approaching the 50 GFLOPS/W goal required for an exascale supercomputer with a power budget at 20 MW. In order to achieve this kind of energy efficiency, heterogeneous computing, i.e. offloading parts of the work to accelerators like graphics processing units (GPUs), plays an important role. It is already known that in terms of performance per watt, GPUs are highly efficient when fast on-chip memory is used, on applications with high parallelism where a single operation is performed on many data elements at a time [13]. Mont Blanc aims to use energy efficient accelerators like the

Application	Description
YALES2	Fluid dynamics
EUTERPE	Fluid dynamics
SPECFEM3D	Seismic wave propagation
MP2C	Multi-particle collisions
BigDFT	Electronic structure
QuantumESPRESSO	Electronic structure
PEPC	Coulomb + gravitational forces
SMMP	Protein folding
ProFASI	Protein folding
COSMO	Meteorological modeling
BQCD	Quantum ChromoDynamics

Table 2.1: List of exascale applications targeted for porting to Mont Blanc (taken from [1])

Library	Description
ATLAS	Automatically Tuned Linear Algebra Software. Provides high performance BLAS (Basic Linear Algebra Subprograms) APIs for C and Fortran [16]
FFTW	Fastest Fourier Transform in The West. For computing Fast Fourier Transforms [17]
HDF5	Hierarchical Data Format version 5. A file format and interfaces for storing large amounts of numerical data [18]
LAPACK	Linear Algebra PACKage. A library providing numerical linear algebra routines, e.g. equation solvers and matrix decompositions [19]
MAGMA	Magma Computational Algebra System. Computer algebra system for solving problems in algebra, number theory, geometry and combinatorics [20]

Table 2.2: List of scientific libraries targeted for porting to Mont Blanc

ARM Mali series of GPUs, which is often used in embedded devices like cell phones [14]. These devices are not designed for HPC and have never been used for such, which is one challenge that the Mont Blanc project aims to address [15].

2.2 OpenMP Super Scalar (OmpSs)

OpenMP (Open Multi-Processing) is a widely used application programming interface for multi-threaded applications in shared memory computers. Compared to many other APIs like pthreads or MPI, OpenMP requires little extra code for adding multiprocessing to the program, which increases the programmer's productivity compared to those other APIs [21]. However, OpenMP is usually implemented for shared-memory multiprocessors, and those implementations cannot be used across nodes in a supercomputer without a virtualization layer like e.g. vSMP [22], or for accelerators like GPUs.

For computing clusters, MPI has been the *de facto* standard programming model, and for GPUs or accelerators, both CUDA, for NVIDIA GPUs, and OpenCL are widely used. However, these programming models involve additional work for the programmer with memory allocation, data movement, device queries, error handling, and so on. As supercomputers grow, this process gets more difficult, which distracts the programmer from the productive work of actually implementing the kernels. There has been some research on programming models to handle heterogeneous architectures, like the Mint programming model and OpenMPC [23][24]. However, those approaches are specialized on only one architecture (NVIDIA GPUs), and they are only efficient on certain types of kernels.

OpenMP Super Scalar (OmpSs for short) is a task-based programming model developed at the Barcelona Supercomputing Center (BSC). The goal of OmpSs is to address the issue of complexity in the development of applications for computing clusters and heterogeneous architectures by providing extensions to OpenMP [25].

2.2.1 Task-based programming

This section will attempt to explain the purpose of and need for task-based programming.

In OpenMP, the programmer would create a parallel region where threads are spawned, and divide the work into tasks or work sharing constructs. For instance, in order to parallelize a SAXPY operation

$$\vec{y} := a\vec{x} + \vec{y} \tag{2.1}$$

the programmer could use one of the kernels in listing 2.1. These two approaches give the same results, although the internals of how the work is divided among the threads are different.

In the first code example the work in the for-loop is simply split between the threads. A default scheduling policy for work sharing is defined by the implementation, though a scheduling policy can be explicitly set.

In the second example, each iteration of the for-loop is defined as an independent *task*. Each task is then assigned to a thread by a task scheduler. The advantage of using tasks is that it is more dynamic; say that you have an implementation of merge sort that should be parallelized. Merge sort is a recursive sorting function, each call to merge sort calls the merge sort function on two halves of the input recursively. This is hard to accomplish using traditional work sharing, since the work is generated recursively. Using tasks, parallelization of such recursive routines is easily expressed by defining each call to merge sort as an independent task.

When writing OmpSs code for SMPs the code for the SAXPY routine is very similar to the task-based OpenMP code, but without the `#pragma omp parallel` block. The reason for this is that in OmpSs, the `#pragma omp parallel` block is implied threads are spawned when the application launches, so the parallel construct is redundant [25]. Listing 2.2 shows the code that runs with OmpSs. As with OpenMP, tasks are created with the `#pragma omp task` construct. These tasks are then scheduled to be exe-

Listing 2.1 Multi-threaded SAXPY in OpenMP

```
void saxpy_worksharing(float* x, float* y, float a, int N) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = y[i]+a*x[i];
    }
}
```

(a) SAXPY in OpenMP with work sharing

```
void saxpy_tasks(float* x, float* y, float a, int N) {
    #pragma omp parallel
    {
        for (int i = 0; i < N; i++) {
            #pragma omp task
            {
                y[i] = y[i]+a*x[i];
            }
        }
    }
}
```

(b) SAXPY in OpenMP with tasks

cuted on one of the threads. Note that work sharing can also be used in OmpSs like in OpenMP, but without the `parallel` directive.

Listing 2.2 Multi-threaded SAXPY in OmpSs

```
void saxpy_ompss(float* x, float* y, float a, int N) {
    for (int i = 0; i < N; i++) {
        #pragma omp task
        {
            y[i] = y[i]+a*x[i];
        }
    }
    //wait for all tasks to complete (synchronization point)
    #pragma omp taskwait
}
```

2.2.2 Heterogeneous capabilities

OmpSs can generate code for symmetric multiprocessors and thus function like traditional OpenMP, but it also supports creating tasks containing code for accelerators using CUDA and OpenCL, as well as Cell processors. The programmer still needs to write device-specific code, but details like data movement, device queries and setup is either hidden from the programmer, or simplified to simple directives.

Instead of having the programmer managing data movement on a device-specific level, OmpSs provides extensions to indicate what data needs to be moved into and out from the device(s) and any dependencies using the directives `inout`, `input` and `output`. For instance, listing 2.3 shows the program outline of SAXPY for both SMPs and OpenCL.

Listing 2.3 Heterogeneous OmpSs program

```c
const int NB = 16;

#pragma omp task inout ([NB] y) input ([NB] x)
void saxpy_block (float* x, float* y, float a)
{
    //SMP specific kernel code for computing y = y + ax for a block of size NB
}

#pragma omp target device (cell) copy_deps implements (saxpy_block)
void saxpy_block_cl (float* x, float* y, float a)
{
    //OpenCL specific kernel code
}

void saxpy(float* x, float* y, float a, int N)
{
    for (int i = 0; i < N; i+= NB)
    {
        //Generate the OmpSs tasks
        saxpy_block(x+i, y+i, a);
    }
    #pragma omp taskwait
}
```

2.2.3 Nanos++ and Mercurium

In the implementation of OmpSs used in this work, developed by BSC, the Nanos++ runtime system is used for task creation and scheduling. Mercurium is a source-to-source translator that takes the source code as input and translates all OmpSs `#pragma omp` directives into calls to the Nanos++ runtime.

2.2.4 OmpSs in Mont Blanc

OmpSs is targeted for use in the Mont Blanc project [1] and has been successfully used in a cluster environment [26]. A computer built on low power devices requires a higher number of computing cores than traditional CPUs in order to achieve the same performance, and the Mont Blanc prototypes will also be using accelerators like GPUs. Both of these points introduce complexities in the code that OmpSs attempts to address. It is as of yet uncertain if OmpSs will be used instead of, or in addition to MPI (Message Passing Interface) in Mont Blanc.

2.2.5 Task scheduling and scheduling algorithms

In OmpSs, threads are spawned as soon as the application starts. The main thread runs serially through the program, and once a task is found, it adds this task to a shared task pool. The inactive worker threads then takes tasks from this pool, puts them into its own private pool of tasks and starts executing them [25]. The worker threads then selects tasks to execute based on the scheduling algorithm that is used.

Some scheduling algorithms utilize work stealing. These algorithms can steal tasks from other thread's private task pools. This can help in instances where the load balance is uneven, but may suffer from less optimal memory locality.

The scheduling algorithm for OmpSs controls which tasks are executed by which thread, which order they are executed in and when they are executed. The choice of the scheduling algorithm can significantly affect the performance of applications in task-based programs [27], both because of memory locality issues, and because of how the tasks are generated. A breadth first algorithm generates all the tasks in the current recursion level before executing them, while a depth-first algorithm does a depth-first traversal of the tasks, and starts executing once the scheduler reaches the bottom-level task.

The default scheduling algorithm in the Nanos++ runtime was found to be depth-first by inspection of the Nanos++ source code. In addition to the default algorithm, distributed breadth-first and work-first will be considered in this thesis. Distributed breadth-first is breadth-first with work stealing. Work-first is depth-first with work stealing.

In chapter 6 we see that choosing the right scheduler for the application has a significant impact on performance.

2.3 Single Instruction, Multiple Data (SIMD)

In recent years, there has been considerable focus on parallelization of software due to the fact that the performance of single core CPUs at some point will reach a wall due to power, instruction level parallelism and memory bandwidth[28][29][30][31]. Increasing the operating frequency of a CPU usually comes with an increase in voltage, and because power is given by [32]

$$P = Cv^2 f \tag{2.2}$$

where C is the capacitance, v is the voltage and f is the frequency. We see that simply increasing the operating frequency, and thus also voltage, is not a sustainable strategy as it would at some point be impractically expensive to keep the chip cool enough.

If frequency stays constant, any increase in performance per processor core must come from an increase in work being done per clock cycle. This can be achieved by having each core execute more instructions per clock cycle, or letting each instruction process more data. The first point is achieved by pipelining the execution of instructions; however, due to data dependencies between instructions, this approach has limits. Introducing vector capabilities is another way of adding more computing power to a CPU core without increasing the frequency. In simplest terms, a CPU capable of vector operations can perform a single operation on multiple data elements at a time, thus is classified as Single Instruction, Multiple Data in Flynn's taxonomy, or SIMD for short.

Although CPUs have traditionally been used for general purpose computing, Graphics Processing Units (GPUs), which traditionally has been purely used for graphics, are now used in scientific applications due to their enormous computing power. GPUs typically get their computing power from vectorization; modern GPU architectures like NVIDIA Fermi uses vector lengths of up to 1024 bits (32 single-precision floating point values) [33], with tens of vector cores performing vector operations in parallel. These GPUs are typically slow on serial operations, so they are limited as a general purpose processor, and also due to the overhead of transferring data to and from the device memory. However, embedded low-power GPUs like the NVIDIA Tegra and ARM Mali GPUs operate with far fewer computing cores [34]. The ARM Mali can also directly access the host's main memory, so the datasets does not need to be explicitly transferred to the device. The NVIDIA Tegra and ARM Mali GPUs are both candidates for use in Mont Blanc [1].

Many modern processor architectures, for instance Intel Sandy Bridge and its predecessors, and the ARM Cortex series of CPUs are capable of doing vector operations on 128-bit vectors. x86 CPUs usually implement Streaming SIMD Extensions (SSE) instructions [35], while ARM CPUs often implement Advanced SIMD Extensions (NEON). In theory, these technologies can give up to four times speedup in execution from non-vectorized code for single precision, or two times speedup for double precision[1]. The newest x86 processors, like the Intel Sandy Bridge and Ivy Bridge, and AMD Bulldozer has support for Advanced Vector Extensions [3], which double the size of the vectors to 256 bits, giving a theoretical speedup of two compared to SSE, and up to eight compared to code not optimized for vector operations for single precision.

2.3.1 Streaming SIMD extensions (SSE)

SSE, designed and introduced by Intel in 1999, is implemented in modern x86-based CPUs today. SSE is based on the older MMX instruction set with 64 bit registers introduced in 1996. The original SSE implementation had most of the basic floating point operations like addition, multiplication, data shuffling, bitwise operations and more for 128 bit registers. SSE2 brought 128-bit integer arithmetics, where SSE1 used the older 64-bit MMX registers for integer operations. SSE3 introduced the capability of working horizontally within a register in only a single instruction, e.g. horizontal reductions ($\{A_0, A_1\}$,$\{B_0, B_1\} \Rightarrow \{A_0 + A_1, B_0 + B_1\}$, whereas vector operations traditionally are performed in a vertical fashion. SSE4 is the latest addition and adds instructions like dot products, min/max, blending, and more.

SSE SIMD instructions are similar to scalar instruction like `fadd`, `fmul`, etc., except they work with larger registers and multiple data elements at once. For instance, the vector instructions `movps` and `addps` corresponds to the scalar instructions for data movement and floating point addition `mov` and `fadd`. The vector instructions are performed on all four (or two, if double precision) elements in the vector in one operation.

In SSE there are 128 bit registers called `xmm0` through `xmm7`; for 64-bit architectures (AMD64, Intel 64), there are eight additional registers `xmm8` through `xmm15`. Typically, the programmer does not need to worry about registers as these are chosen by the compiler.

When creating programs in C that use SSE, there are three typical ways of vectorizing the code:

[1]Note that NEON does not support double precision, so this only applies to SSE

1. Writing inline assembly. This is often hard to read, and gives the compiler little room for optimization, but gives the programmer complete control of the code.

2. Telling the compiler to create vector code automatically. This works well in some cases where it's easy to spot e.g. loops suitable for vectorization, but it is often hard for the compiler to spot code that can be vectorized.

3. Using intrinsics, which is basically wrappers around the assembly for the instructions. This gives the programmer close to total control while also giving the compiler the opportunity of optimizing, e.g. moving instructions around to maximize register use.

Listing 2.4 SAXPY in SSE

```
void saxpy_sse(float* x, float* y, float a, int N) {
    __m128 _a = _mm_set1_ps(a);
    for (int i = 0; i < N; i += 4) {
        __m128 _x = _mm_loadu_ps(&x[i]),
               _y = _mm_loadu_ps(&y[i]);
        _y = _mm_add_ps(_y, _mm_mul_ps(_a, _x));
        _mm_storeu_ps(&y[i]);
    }
}
```

Listing 2.4 shows a simple program that performs SAXPY defined in Equation 2.1 using SSE intrinsics.

2.3.2 Advanced Vector Extensions (AVX)

In 2008, Intel proposed the Advanced Vector Extensions (AVX), as a successor to SSE, with twice as large vector registers for theoretically twice the throughput for vector operations compared to that of SSE. As of writing this thesis, Intel's Sandy Bridge and Ivy Bridge-series, and AMD's Bulldozer processors support AVX. AVX instructions are very similar to SSE, except they work with larger vectors.

AVX registers are extensions of the SSE registers, reusing the 128 bit registers and adding another 128 bits on top of the `xmm*` registers [36]. AVX registers are named the `ymm` registers.

Listing 2.5 SAXPY in AVX: y[i] := ax[i]+y[i]

```
void saxpy_avx(float* x, float* y, float a, int N) {
    __m256 _a = _mm256_set1_ps(a);
    for (int i = 0; i < N; i += 8) {
        __m256 _x = _mm256_loadu_ps(&x[i]),
               _y = _mm256_loadu_ps(&y[i]);
        _y = _mm256_add_ps(_y, _mm256_mul_ps(_a, _x));
        _mm256_storeu_ps(&y[i]);
    }
}
```

Programming with AVX is very similar to programming with SSE: Intrinsics, inline assembly or automatic vectorization can be used. Listing 2.5 shows a programming example with SAXPY, similar to that in section 2.3.1. As we see, the code is nearly identical, just with larger vectors and slightly different names for the intrinsics.

Transition between SSE and AVX

The SSE `xmm` registers is actually the lower half of the `ymm` registers used in AVX. Attempting to write to any of the `xmm` registers after writing to the `ymm` registers will make the CPU store the contents in the `ymm` registers to memory before allowing the `xmm` register to be written, and then writing it back again. This is costly and is called an AVX-SSE transition penalty [36].

Some compilers, e.g. gcc, generates SSE code automatically if it finds a candidate for optimization. This can trigger an AVX to SSE transition penalty, slowing down the execution of the SSE code significantly. To avoid AVX-SSE transition penalties, the instruction `VZEROUPPER` can be invoked to clear the upper halves of the `ymm` registers after use. This makes sure SSE code will not trigger the protection mechanism, as the CPU will know that the register is free. A corresponding intrinsic exists in `gcc` and `icc`, `_mm256_zeroupper()`.

2.3.3 ARM Advanced SIMD Extensions (NEON)

Advanced SIMD Extensions, also called NEON, is the SIMD architecture present in the ARM Cortex CPUs. NEON operates on 64-bit or 128-bit vectors containing elements of size 8, 16 or 32 bits. However, in contrast to SSE, NEON does not have 128-bit registers. Instead, two 64-bit registers are used to hold one 128 bit vector [37]. As a result, most vector arithmetics involving 128-bit vectors are performed separately for each of the 64-bit D registers holding the operands. This is also indicated by the timings for each instruction given at the ARM Infocenter website section for the Cortex-A9 architecture [38]. Consequently, the potential speedup for NEON is lower than that of SSE, because most single-precision floating point operations can only be performed on two floats per cycle.

Listing 2.6 SAXPY in NEON: y[i] := ax[i]+y[i]

```
void saxpy_neon(float* x, float* y, float a, int N) {
    float32x4 _a = vdupq_n_f32(a);
    for (int i = 0; i < N; i += 8) {
        float32x4 _x = vld1q_f32(&x[i]),
                  _y = vld1q_f32(&y[i]);
        _y = vaddq_f32(_y, vmulq_f32(_a, _x));
        vst1q_f32(&y[i]);
    }
}
```

Programming with NEON is similar to programming with AVX or SSE, although with different instructions. Intrinsics, inline assembly, and automatic vectorization are available in `gcc`. An example of SAXPY using NEON intrinsics in `gcc` is given in listing 2.6.

2.4 Energy Efficiency Metrics

This section discusses different metrics for energy efficiency. Energy, power, GFLOPS/W, the energy-delay product and energy-delay squared product are discussed.

2.4.1 Energy and power

Energy is measured in Joules, and power in Joule/second or Watts. Knowing the power dissipation for different optimization methods (e.g. SIMD, or multi-core) reveals information about the energy cost of the different methods.

2.4.2 GFLOPS/W

It is interesting to note that using the terminology from section 1.3, the metric FLOPS/W is equivalent to flops/J, because FLOPS/W = (flops/s)/(J/s) = flops/J. Additionally, if the problem size, and then also # flops, is fixed, FLOPS/W is simply the reciprocal of the total energy spent scaled by a constant, i.e. the # floating point operations. This means that when GFLOPS/W is at a maximum, the energy spent is minimal, and vice versa.

The article "Models and Metrics to Enable Energy-Efficiency Optimizations" [39] discusses different metrics for energy efficiency. For instance, the authors propose the JouleSort benchmark, which measures the number of records that can be sorted per joule. This is similar to the metric flops/Joule, which is equivalent to FLOPS/W used in the Green500 lists as the measure of energy efficiency [11]; both measure the amount of work done per unit of energy, although the quantities are different (# elements sorted, and # flops).

2.4.3 Energy-delay products

Power dissipation is given by equation 2.2, and since v is also dependent on f, reducing the frequency allows us to also reduce the core voltage, and thus the power. However, even though this would likely increase the $FLOPS/W$ value, and reduce the total energy spent, the performance may be much lower. Horowitz et. al [40] proposed the energy-delay product to address this issue, which is defined as $Energy \times Delay$, often abbreviated as EDP. Because both power and performance is related to clock frequency and voltage, the idea is that the EDP would reveal the energy efficiency of the underlying processor design instead of having a metric directly dependent on the clock frequency [39].

Energy-delay squared

Martin et. al proposed a generalization of the energy-delay product on the form $Energy \times Delay^n$ [41]. Here, n can be chosen emphasize performance in the energy-delay product. $n = 2$ is discussed thoroughly in the article as a reasonable choice for many applications. The article argues that the traditional EDP is not sufficient to compare implementations when voltage scaling is involved. It gives an example with a log and linear implementation of a comparator, where the energy usage of the log comparator is higher than the linear one if the voltages are the same, giving a higher EDP even if the delay is lower. However, when the voltage of the log comparator is scaled so that the delay matches that of the linear one, the energy spent is lower, and thus it is revealed that the log implementation is better. The ED^2 metric reveals this even if the voltages are the same. ED^2 is sometimes abbreviated to as EDD.

Technology scaling: Process-normalized EDP and EDD

If the transistor size due to the manufacturing process is scaled by a factor of λ, under ideal conditions, the energy-delay product is scaled by λ^4 [40]. In the paper "Energy dissipation in general purpose microprocessors" [42], the authors argue that because the processor speed is limited by other factors like the memory system, and that there is a threshold voltage that cannot be subsided in order for the processor to function, the EDP scaling factor likely lies between λ^2 and λ^3. The process-normalized EDP can then be given as

$$EDP_{normalized} = \frac{Energy \times Delay}{\lambda^s} \tag{2.3}$$

where s is the scaling factor and $2 \leq s \leq 4$. In [42], λ^2 is used as the scaling factor in the processor comparisons, which is considered the lower-bound for the scaling factor [40].

Horowitz et. al. [40] state that the delay and energy is each reduced by a factor of λ due to reduced capacitance alone if the transistors are scaled with a factor λ. Although no papers defining a reasonable normalization factor for EDD was found, we will make an attempt to derive one. Since the EDD is simply EDP $\times$ $Delay$ and if the normalization factor for the EDP is λ^s, we will use the normalization factor λ^{s+1}, as it simply counts the factor due to the delay twice. This gives the *normalized EDD*

$$EDD_{normalized} = \frac{Energy \times Delay^2}{\lambda^{s+1}} \tag{2.4}$$

2.5 Related Work

In this section we present a selection of research papers, white papers and online resources related to the thesis.

2.5.1 The Energy Efficiency of CMP vs SMT for Multimedia Workloads

Sasanka et. al compares the energy efficiency of chip multiprocessors (CMP) and single-core processors utilizing simultaneous multi-threading (SMT) when applied to multimedia applications [43]. The authors models a dual-core and quad-core CMP, and two different SMT processors supporting two and four simultaneous threads, respectively. They use the normalized energy per instruction as the energy metric, and time per instruction for performance. CMPs are found to be more energy efficient; the dual-core use 9% less energy on average and the quad core use 39% less energy compared to SMTs with two and four threads, respectively.

A hybrid model with SMTs as cores in a CMP is also modelled, which is shown to use 11% more energy on average compared to the CMP. Intel multi-core processors with hyper-threading uses this hybrid model.

2.5.2 Energy per Instruction Trends in Intel®Microprocessors

Grochowski et. al presents an overview of the trend of the energy consumption per instruction (EPI) in Intel processors from the 486 to the Intel Core Duo family [44]. The performance and energy is process-normalized in order to remove the factor of transistor size. The authors show a strong increase in the EPI up to the Pentium 4 Willamette processor with an EPI of $48nJ$, and then almost an equally strong decrease down to an EPI of $11nJ$ for the Core Duo Yonah. As a comparison, the 486 processor is stated to have an EPI of $10nJ$.

2.5.3 Evaluation of OpenMP Task Scheduling Strategies

A part of this thesis is dedicated to finding a good scheduling algorithm for the benchmarked applications. A. Duran et.al shows that in many applications, the work-first algorithm works best [27]. The work-first algorithm attempts to follow the serial execution pattern of the application, and thus exploits memory locality that would be present as if the application were to be run serially. Breadth-first is another scheduling scheme where all tasks for the current recursion level is put in the task pool before execution of the tasks on the next recursion level commences. The authors finds that breadth-first algorithm is inferior to work-first in most of their benchmarks.

In this project, a different set of applications are benchmarked, and using OmpSs instead of OpenMP. The results in [27] are relevant and used as a guideline for what schedulers should be explored.

2.5.4 Parallelization of Black-Scholes and dense matrix-matrix multiply using OmpSs

A. Duran et.al "OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures" [25] presents results from performance benchmarks for both Black-Scholes and matrix-matrix multiply where OmpSs is used. The CPU-version of the matrix-matrix multiply algorithm performs at 4 GFLOPS with four cores for most problem sizes. For Black-Scholes, no absolute performance numbers are presented; instead, the paper presents speedups from increasing the number of threads. The speedups presented are against the serial version that does not use OmpSs. One thread using OmpSs tasks achieve 2.6x speedup due to the SIMD OpenCL kernel that is generated. For four threads, the speedup is 10, or 3.84 compared to the single threaded OmpSs implementation. The paper does not present energy efficiency results, which is the goal of this thesis.

2.5.5 Benchmarking of FFTW

The FFTW library is extensively benchmarked in terms of performance on a variety of platforms [45][17][46]. The results show that FFTW has a definite peak around where the problem fits in the faster caches of the CPU, after which the performance drops. These papers presents impressive performance results for single-threaded execution versus other popular FFT algorithms, however no results was found on power consumption on different devices. Also, no benchmark was presented for newer processors using more than one core.

2.5.6 Energy efficiency of IRAM architectures

Fromm et. al. presents energy efficiency models for IRAM architectures, where DRAM is integrated on the processor chip [47]. Because the DRAM is present on-chip, the authors argue that the energy consumption is reduced compared to using a traditional off-chip DRAM memory system. The authors found that the energy consumption of IRAM systems is significantly lower than systems using a conventional memory hierarchy. The paper uses the metric energy per instruction metric in the results.

2.5.7 Green500

Green500 is a ranking website for energy-efficient supercomputers [11]. The site ranks the machines listed in the Top500 list by energy efficiency using GFLOPS/W, where the performance is measured using LINPACK, and the power is measured off the same benchmark. The Green500 Run Rules list specific guidelines and rules for how the measurements must be made [48]. As of November 2011, the top ranking computer is the IBM BlueGene/Q, Power BQC with 2.02 GFLOPS/W.

2.5.8 Energy-saving mobile processor architectures

Goulding-Hotta et. al. presents an alternative to the traditional general purpose processor used in devices running the Android operating system [49]. The authors estimate a 91% reduction in energy consumption by using specialized circuits for commonly used functionality, like Fast Fourier Transforms or JPEG decompression. The reductions are possible because these chips do not need a instruction fetch/decode unit, instruction cache, register file or a generalized data path.

Chapter 3

Application Kernels

This chapter gives an overview of the application kernels that was chosen for benchmarking in this thesis.

Section 3.1 describes the Black-Scholes model and the application kernel, while section 3.2 gives a short description of fast Fourier transforms and their applications, as well as the FFTW library. Section 3.3 gives a brief overview of the matrix multiplication kernel.

3.1 Black-Scholes (BSOP)

In this section the Black-Scholes model is explained. Black-Scholes were chosen because it has previously been benchmarked as an OmpSs application, and is part of the PARSEC Benchmark Suite for shared memory computers [50]. The Black-Scholes model is a mathematical model of a financial market, describing the price of an option over time [51]. An option in the financial sense of the word is a contract of selling and buying some underlying asset where one of the parties are obligated to sell/buy the asset, while the other party is not. This has financial value, as the party that is not obligated to do anything may choose whatever is most beneficial for him or her, even if that involves a financial loss for the other party. This price is what the Black-Scholes model attempts to model. The Black-Scholes equation is the partial differential equation given in equation 3.1.

$$\frac{\delta V}{\delta t} + \frac{1}{2}\sigma^2 S^2 \frac{\delta^2 V}{\delta S^2} + rS\frac{\delta^2 V}{\delta S} - rV = 0 \tag{3.1}$$

where

S is the price of the underlying stock.

σ is the volatility of the stock's returns.

$V(S,t)$ is the price of a option.

t is the time, in years.

r is the annual risk-free interest rate.

From this differential equation, the Black-Scholes formula can be derived, which describes the price of European-style call and put options. A call option is an option where the seller is required to sell if the buyer wants to buy, but the buyer is free to not buy the asset. A put option is the opposite, where the buyer is required to buy if the seller wants to sell the asset, but the seller may choose wether or not to sell. European style options are options that must be exercised at the time of maturity, i.e. the agreed point in time for the trade. As a contrast, American style options can be sold/bought at any point in time up until the time of maturity. The Black-Scholes formula is derived by setting the appropriate boundary conditions, and is shown in equations 3.2 through 3.5. The derivation of the formula is outside of the scope of this thesis but can be found in [51].

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)} \tag{3.2}$$

$$P(S,t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S \tag{3.3}$$

$$d_1 = \frac{\ln\frac{S}{K} + (r + \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} \tag{3.4}$$

$$d_2 = \frac{\ln\frac{S}{K} + (r - \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} = d_1 - \sigma\sqrt{T-t} \tag{3.5}$$

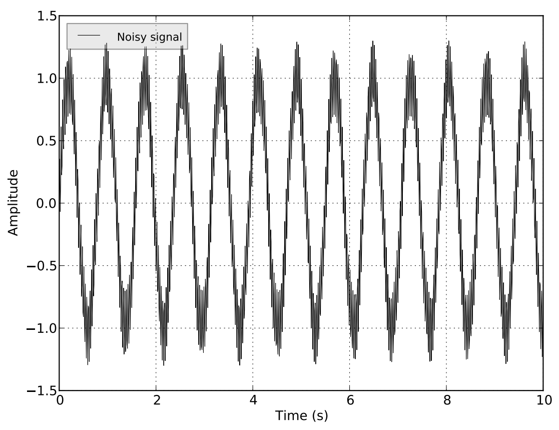
Here, the function N is the cummulative distribution function of the standard normal distribution, $T - t$ is the time to maturity (the option matures at time T), and K is the strike price. The functions C and P is the call and put option prices, respectively. The rest of the symbols are explained after equation 3.1.

The formula can be trivially evaluated in parallel by computing the value of the options at each point of time t in parallel. The implementation of the Black-Scholes kernel is more thoroughly explained in chapter 4.

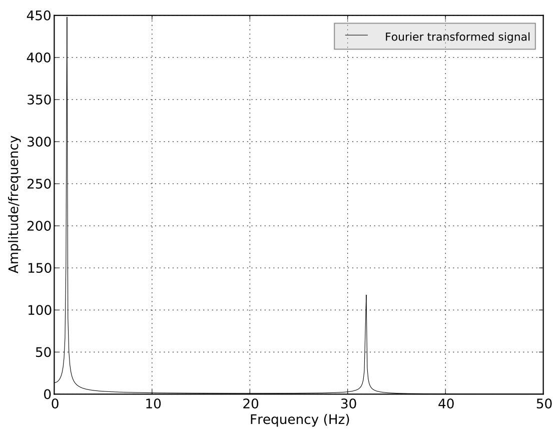
3.2 Fastest Fourier Transform in the West (FFTW)

This section describes the FFTW library that is used for benchmarking in this thesis. FFTW is a relevant library for many scientific applications, and is one of the libraries targeted for porting to Mont-Blanc. There has been considerable research in the area of discrete Fourier transforms, especially fast Fourier transforms (FFT). FFT is a very important class of algorithm, and its applications are seemingly endless: filtering (e.g. filtering noise, or doing high/low pass filtering) in both signal processing, image processing [52]; solving linear equations [53]; multiplication of large numbers [54], and more.

The discrete Fourier transform transforms a signal from the time or space domain to the frequency domain; this allows for a different view of the signal, for instance, instead of seeing periodic noise in an audio signal, we would see a peak in the frequency spectrum at the frequency of the signal. For instance, consider the signal in figure 3.1a. Here there is a signal with high-frequency noise added. If we apply a Fourier transformation on this signal and plot the amplitude of each frequency, we get the curve in figure 3.1b. We see that there are two peaks: One in the low frequency area (near zero on the x-axis, more specifically at about $1.27Hz$), as well as one further out, at about $32Hz$. That is the peak for the high-frequency noise.



(a) Signal with respect to time



(b) Fourier spectrum

Figure 3.1: Sine wave with high frequency noise

Setting the values of the high-frequency interval (roughly $32Hz \pm 10Hz$) to zero for the transformed curve, and then performing an inverse transform yields the noise-reduced signal in figure 3.2.

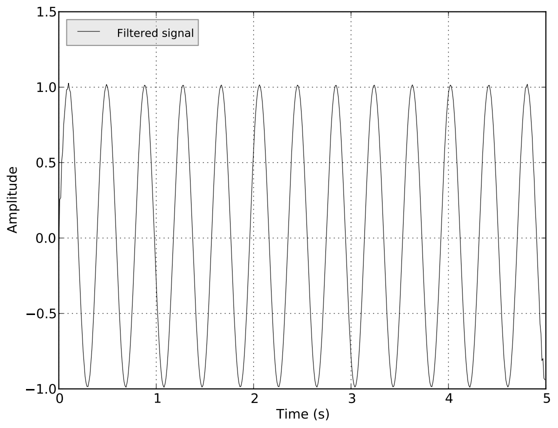


Figure 3.2: Sine wave with noise filtered out

Computing the discrete Fourier transform naively takes $O(n^2)$ time for n elements [55], which is too slow to be practical for many applications. However, the Fast Fourier Transforms is a class of algorithms designed to find the discrete Fourier transform in only $O(n \log n)$ time [55]. It works by subdividing the problem into smaller pieces, e.g. splitting the input in half, and computing two smaller DFTs, which may be split further. We will not go deeper into the theory of FFTs as that is outside of the scope of this work.

The FFTW library is a library made for efficient computation of the Fourier transform and its inverse. It is regarded as one of the most efficient FFT libraries as it automatically tunes the algorithm to the machine it's run on, by first creating a plan of execution for the given problem, and then executing that plan. The plan is created by using some heuristics to adapt execution to the current system (e.g. querying the cache sizes), and also executing many different plans to find the best one. The plans may be saved to disk for reuse on other problems. Optionally, the measurement part may be left out to save time on creating the plan, yielding a less optimal plan for execution.

The FFTW plans contain information about how to most efficiently recursively subdivide the problem, and also how to most efficiently solve the base cases. The base cases are sufficiently small problems that can be solved directly using a series of small highly optimized code segments called *codelets*. When the plan is executed, the problem is subdivided according to the plan. Then, codelets are executed for each of the base cases.

3.3 Dense General Matrix-Matrix Multiplication and ATLAS

Matrix multiplication is a fundamental linear algebra operation which is used as a building block in many scientific applications.. It is also a popular application for benchmarking due to its very high floating point operations-to-memory access ratio in the blocked implementation of this algorithm. Matrix-matrix multiplication is a level 3 BLAS (Basic Linear Algebra Subprograms) routine, and is generalized to

$$C = \alpha AB + \beta C$$

where A, B and C are matrices with dimensions $m \times k$, $k \times n$ and $m \times n$, respectively. The naive way of computing AB is simply calculating

$$C_{ij} = \sum_{l=1}^{k} A_{il} B_{lj} \tag{3.6}$$

which can be implemented in three simple nested for-loops. Equation 3.6 implies $2mnk$ floating point operations in total (k additions of k multiplications, mn times) for computing C_{ij} for every i and j. However, the naive algorithm typically performs poorly on large matrices because it does not exploit cache locality, and thus gets low reuse of data for matrices that are too big to fit in cache. Instead, matrix multiplication can be implemented in a blocked fashion, where the matrices are divided into blocks that themselves fit in cache, and then perform block-wise multiplication. Blocked matrix-matrix multiply gives a much better cache reuse, and consequently better performance.

Without loss of generality, assume that A, B and C are square, with dimensions $n \times n$. Let the blocks have dimensions $b \times b$, and let b and n be chosen so that b divides n evenly. Define N to be n/b, i.e. the number of block rows and columns in M, and let M_{IJ} denote the Jth block column in the Ith block row of M, where M is A, B or C. Then blocked matrix-matrix multiplication can be defined as

$$C_{IJ} = \sum_{L=1}^{N} A_{IK} B_{LJ} \tag{3.7}$$

Each block-block multiplication $A_{IK}B_{LJ}$ can be computed using the naive matrix-matrix multiplication in equation 3.6. It is important to note that the total number of floating point operations performed in the blocked algorithm is the same as in the naive calculations, i.e. $2N^3$. This algorithm is generalized to non-square matrices where $m \neq k \neq n$ by choosing the block size $a \times b$ for A and $b \times c$ for B.

With blocks of size $b \times b$, each element in A and B may have to be loaded N/b times from potentially slow memory, as opposed to N times without blocking. The total number of elements in A and B are $2N^2$, so the total number of loads from slow memory, ignoring cache line sizes, is $2N^3$ for the unblocked algorithm, and $2N^3/b$ for the blocked algorithm. For the naive algorithm, the flops per memory access ratio is 1, while for the blocked algorithm, b flops can be done per slow memory access. In other words, the amount of reuse of each data element is b; the larger blocks, the more reuse. This can be utilized to have a hierarchy of blocked matrices, where the block sizes in each are made to fit each level of cache, giving a highly efficient matrix multiplication algorithm.

ATLAS is an auto-tuned BLAS implementation, that automatically selects the best performing kernels for the different BLAS routines, including matrix multiplication [16]. The parameters for the kernels, like the block size, are chosen according to the physical parameters for the machine like cache size, and several different kernels are tried out to find the best one for the architecture. This is all done during compile time. As a result of that auto-tuning, ATLAS is known to have a high performance on a wide range of architectures.

Chapter 4

Implementation

4.1 Energy Measurement Algorithms

In this section, we present the algorithms used for measuring the consumed energy for both Sandy Bridge and the ARM Cortex-A9 MPCore CPU. For Sandy Bridge, the consumed energy can simply be read out from a register, while on ARM, the power dissipation must be measured at regular intervals before the energy consumption is found by numerical integration.

4.1.1 Sandy Bridge

For energy measurements on the Sandy Bridge, code based on the utility program `rdmsr` is used, which reads the fields for energy consumption in the Machine State Register (MSR), and gives as output the values for energy consumed for both power plane 0 (PP0), power plane 1 (PP1) and the package. Power plane 0 typically refers to the CPU cores with their respective caches [56, Vol 3B, sec. 14.7.4 PP0/PP1 RAPL Domains], while the package energy refers to the whole processor package, i.e. all cores plus the caches and any other on-chip controllers like for instance memory controllers [56, Vol 3A, sec. 8.9.1 Hierarchical Mapping of Shared Resources].

The measurement code reads the energy consumption values from the MSR registers, using the RAPL (Running Average Power Limit) interface described in the Intel Architectures Software Developer Manual [56, Vol 3B, sec. 14.7.1 RAPL Interfaces]. These registers describe the units and granularity of the values in the registers, and also contains the values themselves.

The most relevant MSRs for this project is read using the library is the `MSR_-PKG/PP0/PP1_ENERGY_STATUS`, in conjunction with the `MSR_RAPL_POWER_UNIT` registers. `MSR_PKG_ENERGY_STATUS` bits 31:0 contains the value for energy consumed for the whole processor package. `MSR_PP0_ENERGY_STATUS` and `MSR_PP1_ENERGY_-STATUS` is similar, but for only the components directly related to each of the CPU cores like the L1/L2 caches and the cores themselves, and for power plane 1, respectively. The values are stored as integers and are specified to have a wrap-around time of about

60 seconds on high load [56, Vol 3B, sec. 14.7.3 Package RAPL Domains].

`MSR_RAPL_POWER_UNIT` describes the units of time, energy and power used in the values. Most relevant are bits 12:8 which describes the units, or granularity, of the `MSR_PKG_ENERGY_STATUS` register. The default value is $0b10000 = 16$, and the unit is computed as $u = 2^{-}ESU$, which gives a default granularity of `MSR_PKG/PP0/PP1_-ENERGY_STATUS` of $15.3\mu J$ [56, Vol 3B, sec. 14.7.1 RAPL Interfaces]. In other words, in order to get the energy consumed in Joules, one has to multiply the value in the `MSR_PKG_ENERGY_STATUS` MSR by $15.3 \cdot 1000^{-2}$.

On Linux systems, the MSR registers are made available through the device file `/dev/cpu/cpu0/msr` after loading the `msr` kernel module.

4.1.2 ARM Cortex-A9 MPCore

On the ARM Cortex-A9, the current power dissipation of the CPU can be made available through the three 32-bit system configuration registers, called the `SYS_CFG` registers `SYS_CFGCTRL`, `SYS_CFGDATA` and `SYS_CFGSTAT`, for control, data and status, respectively [57]. `SYS_CFGCTRL` is written to in order to either set some system parameter; for instance, the frequency of the oscillators in the system can be set through this register, or the system can be rebooted or shut down. Additionally, `SYS_CFGCTRL` is written to in order to retrieve a value or measurement such as power dissipation, temperature, voltage, and more. If a read request was put into `SYS_CFGCTRL`, the requested value is put into `SYS_CFGDATA` if the request is valid and it succeeds. In `SYS_CFGSTAT` bit 0 indicates completion of the read or write request, and bit 1 indicates an error in fulfilling the request.

Bits	**Description**
31	Start bit. If set to 1, the request are to be initiated as soon as the register is read.
30	Read/write bit. If set to 1 (0), initiate a write (read) request.
29:26	Daughter board configuration controller (DCC).
25:20	Function. Describes the function or value to be read or written. For power, the value is 12. A full overview of functions can be found at [57].
17:16	Site. Which board the device resides on. Motherboard is 0, daughter boards are 1 and 2.
15:12	Stack position, if multiple daughter boards are stacked on the given site.
11:0	Device number. Which device to be read, on the given board.

Table 4.1: Overview of the `SYS_CFGCTRL` register

An overview of the important bits of `SYS_CFGCTRL` is shown in table 4.1. Note that bits 19:18 are undefined and is not listed. Based on this overview, a routine to sample power can be described. First, write the appropriate value to `SYS_CFGCTRL`. Then, wait until the complete bit in `SYS_CFGSTAT` is set. Finally, read the value from `SYS_CFGDATA`.

For the hardware used for benchmarking, the daughter board controller ID is 0, site is 1 and stack position is 0. For power measurement, there are two devices available: Device 0 is the PL310 L2 cache controller and SRAM, and device 1 is the Cortex-A9 CPU cores [58]. In order to read the power from the CPU cores, the value `0x80C10001` is written to `SYS_CFGCTRL`, or in binary,

```
Start R/W DCC  Funct. Undef Site Spos Device
1     0   0000 001100 00    01   0000 000000000001
```

For the L2 cache controller and SRAM the value is `0x80C10000`.

On Linux-systems, the `SYS_CFG` registers can be accessed by memory mapping the device `/dev/mem` using the `mmap` function in `sys/mman.h`. The address for the `SYS_CFG` registers is the base address for system registers, which is given by the memory map for the board [59], plus the offset for the `SYS_CFG` registers [60]. For the board used in this thesis, the base address is `0x10000000`, and the offsets are `0xA0`, `0xA4` and `0xA8` for `SYS_CFGDATA`, `SYS_CFGCTRL` and `SYS_CFGSTAT` respectively. The full algorithm for sampling the power is given in algorithm 1.

Algorithm 1 Basic power sampling procedure

```
  // Procedure shows sampling of device 1 (A9 cores).
  // The procedure for sampling device 0 is the same.
  procedure SAMPLE-POWER
      Write 0x80C10001 to memory address 0x100000A4
      Sleep for $300\mu s$
      while Value at address 0x100000A8 bitwise-AND 0x1 is 0 do
          Sleep for $50\mu s$
      end while
      return Value at address 0x100000A0
  end procedure
```

In order to measure the energy consumption of a running application, sampling is done in a separate thread at regular intervals. After the control register has been written to with a read request for power, it takes a certain number of cycles before the data is available in the data register. The status register indicates when the request has been completed, and the data is ready. To minimize the impact of the sampling thread on the performance of the application, the thread sleeps after the control register is written before the data is ready instead of spinning on the status register. The sleep time was found by measuring the time from request to completion, and verified by counting the number of spins in the while loop and checking that the count was zero. The latency was found to be approximately $300\mu s$, although with slight variations.

The total energy consumption is given as

$$E = \int_0^T P(t)dt \tag{4.1}$$

where T is the end time of the application, and $P(t)$ is the power at any given time t. The power is sampled at regular intervals and integrated numerically in order to get the total energy consumption. Using the trapezoidal rule for numerical integration, the total energy is approximated as:

$$\int_0^T P(t)dt \approx \sum_{i=0}^{n-1}(t_{i+1} - t_i)\frac{P(t_{i+1}) + P(t_i)}{2} \tag{4.2}$$

where n is the number of samples. In practice, the time intervals $(t_{i+1} - t_i)$ are not constant and the number of samples n is not known because reading out the energy takes time. Additionally, thread scheduling introduces variance in the interval lengths as the sampling thread may be context switched in and out by the operating system. Therefore the interval between each measurement must be used instead of a constant interval. The full energy measurement algorithm using sampling is presented in algorithm 2.

Algorithm 2 Energy measurement using power sampling

```
Global: $t_{end}$; initially, $t_{end} = \infty$ // Indicates when the benchmark has ended
procedure MEASURE-ENERGY
    $p_0 := -1, p := -1, t_0 :=$NOW(), $E := 0$
    while Benchmark is running do
        SLEEP() // Sleep until the next sampling
        $p :=$SAMPLE-POWER()
        // The test may have ended while sampling.
        // If so, use the end time instead of the current time.
        $t :=$MIN($Now(), t_{end}$)
        if $p_0 < 0$ then
            $p_0 := p$
        end if
        $E := E + (t - t_0)(p + p_0)/2$ // Numerical integration step
        $p_0 := p, t_0 := t$
    end while
    return $E$
end procedure
```

4.2 Porting of SSE log and exp to AVX and NEON

In Black-Scholes there is extensive use of both natural logarithms and the exponential function. Unfortunately, the standard C and C++ libraries do not include vectorized versions of these functions, though they do include a scalar implementation. Computing the exponential and natural logarithms for every element in the vectors in a vectorized kernel would be a significant bottleneck. This is because not only are the functions evaluated serially, but data must also be transferred between the vector registers and general purpose registers.

The Cephes Mathematical Library provides an extensive number of mathematical functions relevant to scientific applications [61]. The library is freely distributable and written in C. Julian Pommier has developed an SSE and SSE2 version of the `sin`, `cos` , `log` and `exp` functions in this library called `sin_ps`, `cos_ps`, `log_ps` and `exp_ps`, respectively, and made his work freely available [62]. Parts of Pommier's code were adapted to use AVX and NEON, specifically the `exp_ps` and `log_ps` functions.

4.2.1 Porting to AVX

Porting of the SSE code for `log` and `exp` to AVX is mostly trivial. First, variables of types `__m128` and `__m128i` was made into `__m256` and `__m256i`. Then the intrinsics for 128 bit SSE instructions were replaced with the 256 bit AVX instructions, where possible. Most 128 bit intrinsics used in `log_ps` and `exp_ps` do have a 256 bit equivalent; e.g. `_mm_add_ps` has a 256 bit equivalent called `_mm256_add_ps`.

However, element-wise bit shifting (e.g. instructions `PSRLD` (right shift) and `PSLLD` (left shift) in SSE, with corresponding intrinsic `_mm_srli_epi32` and `_mm_slli_epi32`) are not introduced before AVX2 [63, appendix A], which is first implemented in Intel's Haswell architecture, set to be released in 2013 [64]. Also, 256 bit integer arithmetic are missing from AVX1. In this project, bit shift operations and integer arithmetic were done in SSE. Listings 4.1 and 4.2 show the conversion.

Listing 4.1 Missing 256 bit instructions workaround in logarithm function

```
// Cast the float vector x to ints, then shift each element right by 23 bits
emm0 = _mm_srli_epi32(_mm_castps_si128(x), 23);
// Then do one subtraction
emm0 = _mm_sub_epi32(emm0, *(__m128i*)_pi32_0x7f);
```

(a) Original 128 bit code in log_ps()

```
// - Extract the lower (offset 0) 128 bits from the 256 bits register
// - Cast the floats to ints
// - Shift right by 23 bits
__m128i emm01 = _mm_srli_epi32(_mm_castps_si128(_mm256_extractf128_ps(x, 0)),23);
// Do the same for the upper (offset 1) 128 bits
__m128i emm02 = _mm_srli_epi32(_mm_castps_si128(_mm256_extractf128_ps(x, 1)),23);
// SSE integer addition
emm01 = _mm_sub_epi32(emm01, *(v4si*)_pi32_0x7f);
emm02 = _mm_sub_epi32(emm02, *(v4si*)_pi32_0x7f);
//Insert the 128 bit vectors into the lower and upper part of the 256 bit vector,
    respectively
emm0 = _mm256_insertf128_si256(emm0, emm01, 0);
emm0 = _mm256_insertf128_si256(emm0, emm02, 1);
```

(b) Workaround for 256 bit in log256_ps()

4.2.2 Porting to NEON

SSE and NEON intrinsics use different datatypes in their implementations. SSE uses the data types `__m128` and `__m128i` for 128-bit floating point and integer vectors, respectively. NEON uses data types named `[type][P]x[N]` where *type* describes the data type (e.g. float, int, uint), P indicates how many bits there are per elements, and N is how many elements there are. So the datatype that holds four 32-bit floating point numbers is called `float32x4`, a data type that holds two unsigned integers are called `uint32x2`, and so on.

Most of the intrinsics are directly mapped to NEON instructions. For instance, the intrinsic `float32x4 C = vaddq_f32(A, B);` is mapped directly to `vadd.f32 qC, qA , qB`, where `qA`, `qB` and `qC` are 128-bit vector registers. This, however, causes intrinsics that use/return different data types to be incompatible, even though they are fully com-

Listing 4.2 Missing 256 bit instructions workaround in exponential function

```
// Addition
emm0 = _mm_add_epi32(emm0, *(__m128*)_pi32_0x7f);
// Shift each 32 bit element left by 23 bits
emm0 = _mm_slli_epi32(emm0, 23);
```

(a) Original 128 bit code in exp_ps()

```
// Extract 128 bit vectors from a 256 bit vector, do one addition and shift each 32 bit
     element left by 23 bits
__m128i emm01 = _mm_slli_epi32(_mm_add_epi32(_mm256_extractf128_si256(emm0, 0), *(
    __m128i*)_pi32_0x7f), 23),
        emm02 = _mm_slli_epi32(_mm_add_epi32(_mm256_extractf128_si256(emm0, 1), *(
    __m128i*)_pi32_0x7f), 23);

// Insert both 128 bit vectors into one 256 bit vector
emm0 = _mm256_insertf128_si256(emm0, emm01, 0);
emm0 = _mm256_insertf128_si256(emm0, emm02, 1);
```

(b) Workaround for 256 bit in exp256_ps()

patible in pure assembly. An example of this is comparison. Comparison instructions in SSE, AVX and NEON simply set all the bits of the elements that evaluates to true to one in the result vector, and the rest to zero. Sometimes, statements like `if (a > b) a ++;` are performed. In NEON assembly, assuming `a` is in `q0`, and `b` is in `q1`, this can be written as in listing 4.3.

Listing 4.3 `if (a > b) a++;` with NEON assembly

```
vcgt.f32 q2, q0, q1; //compare, and put result in q2
vmov.f32 q3, 1.0;    //put the floating point value 1.0 in q3
vand     q2, q2, q3; //do bitwise-AND of q2 and q3 (q2 will then have the value 1.0 in
    elements where the comparison were true, and 0 otherwise)
vadd.f32 q0, q1, q2; //add q2 (where each element is either 1.0 or 0) to q0
```

However, the intrinsics for comparison and logical bitwise operations only returns unsigned integers. This means that the result must be reinterpreted as a floating point data type before it is added to `a`. This is done using the `vreinterpretq_f32_u32(A)` intrinsic, which reinterprets the bits in A, and returns a 128-bit vector containing four floating point values. This may also be an issue for SSE in some circumstances, in that instruction set the comparison and bitwise logical operations has intrinsics for both integer and floating point types. For AVX, only intrinsics for floating point types exist for comparison and bitwise logical operations. Listing 4.4 demonstrates how the assembly in listing 4.3 is written using intrinsics. Since the NEON registers are typeless, the reinterpretation intrinsics should be free, as they are only needed for the C API.

Other than the required reinterpretation casts for some of the intrinsics, the SSE intrinsics in the exponential and logarithm codes could more or less be translated directly with corresponding NEON intrinsics.

Listing 4.4 `if (a > b) a++;` with NEON intrinsics

```
uint32x4 mask = vcgtq_f32(a,b); //perform the comparison, store result in "mask"
mask = vandq_u32(mask, vreinterpretq_u32_f32(vdupq_n_f32(1.0f))); //duplicate the value
     1.0 to all elements of a floating point vector, and reinterpret it as an unsigned
    integer for the bitwise-AND
a = vaddq_f32(a, vreinterpretq_f32_u32(mask)); //reinterpret the mask as a floating
    point vector, and add it to a
```

4.3 Black-Scholes

An OmpSs implementation of Black-Scholes was provided by BSC which did not contain any vectorized code. The pseudo-code of the kernel in this implementation is given in algorithm 3. The procedure Black-Scholes evaluates the Black-Scholes formula in order to find the price of every call and put option. Each evaluation is done independently from every other, so this function has a considerable potential for parallelization, both through vectorization and multi-threading.

In Black-Scholes, the processing of every element is independent, and thus is an excellent candidate for parallelization.

4.3.1 Vectorization

Three vectorized versions of Black-Scholes were developed: one for SSE, AVX and NEON. Instead of passing one element to the Black-Scholes-evaluation function, four or eight are passed for SSE or AVX, respectively. In practice, the address to the Li-th element of each input-array is passed, where L is the vector length (four for SSE and NEON, eight for AVX) and i is the loop counter. Listing 4.5 shows a simplified version of the outer loop. Here `vector_width` is set elsewhere, and is eight if compiling for AVX, four for SSE or NEON, and one otherwise. The source code for the full OmpSs-enabled outer loop can be found in listing E.1 in appendix E.

The input arguments to the Black-Scholes formula evaluation function, and to the function `N`, is read into vector registers with `_{mm,mm256}_loadu_ps` for SSE/AVX, and `vld1q_f32` for NEON; e.g. the argument `T` is loaded using `__m256 _T = _mm256_loadu_ps (T)` in AVX, where `_T` is a 256 bit vector, and `T` is a pointer to a sequence of at least eight 32 bit floats in the `T` array (see algorithm 3). A corresponding load for NEON would be `float32x4 _T = vld1q(T).`

The Black-Scholes evaluation function was modified to use vector instructions instead of scalar arithmetic operators. Multiplication `a = b * c` is replaced by `va = _{mm,mm256}_mul_ps(vb, vc)` for SSE and AVX, and similarly `va = vaddq_f32(vb , vc)` for NEON, where a, b and c are scalars and va, vb and vc are vectors. Addition and subtraction are replaced with `va = _{mm,mm256}_{add,sub}_ps(vb,vc)` in SSE/AVX and `vsubq_f32` in NEON in a similar fashion. In SSE and AVX, division and square root is replaced with `va = _{mm,mm256}_div_ps(vb, vc)` and `va = _{mm ,mm256}_sqrt_ps(vb)`, respectively.

IEEE 754 compliant division and square root are not implemented in NEON. In-

Algorithm 3 Reference Black-Scholes algorithm

```
procedure N(x)// Standard normal distribution CDF
    $k := (1 + 0.2316419 \cdot |x|)^{-1}$
    $a := -1.821255978 + 1.330274429k$
    $a := ak + 1.781477937$
    $a := ak - 0.356563782$
    $a := ak + 0.319381530$
    $a := ak$
    $n := \frac{1}{2\pi} e^{-0.5|x|^2}$
    if $x < 0$ then
        return $an$
    else
        return $1 - an$
    end if
end procedure

procedure EVAL-BLACK-SCHOLES($C, S, K, r, \sigma, T$)
    $d_1 := (\ln \frac{S}{K} + (r + \frac{1}{2}\sigma^2)T)/(\sigma\sqrt{T})$
    $d_2 := d_1 - \sigma\sqrt{T}$
    $E := e^{-rT}$
    $N_1 :=$N($d_1$)
    $N_2 :=$N($d_2$)
    $c := N_1 S - N_2 K E$
    $p := (1 - N_2)KE - (1 - N_1)S$
    if $C$ then
        $A = c$
    else
        $A = p$
    end if
    return $A$
end procedure

procedure BLACK-SCHOLES($C, S, K, r, \sigma, T, n$)
    $A :=$ Empty array for results
    for $i := 0 \to n$ do
        $A_i :=$EVAL-BLACK-SCHOLES($C_i$,$S_i$,$K_i$,$r_i$,$\sigma_i$,$T_i$)
    end for
    return $A$
end procedure
```

C: Flags indicating whether we want a put or call option (`true` if call).

S: Spot prices of underlying asset

K: Strike prices

r: Risk free (annual) rates

σ: Volatility of returns of the underlying asset

T: Time to maturity

n: Problem size

Listing 4.5 Outer for loop of Black-Scholes (simplified)

```
for (i=0; i<array_size; i+=vector_width)
{
    #if defined(AVX)
    bsop_avx(&answer_fptr[i], &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &r_fptr[i], &
    sigma_fptr[i], &T_fptr[i]);
    _mm256_zeroupper();
    #elif defined(SSE)
    bsop_sse(&answer_fptr[i], &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &r_fptr[i], &
    sigma_fptr[i], &T_fptr[i]);
    #elif defined(NEON_INTRIN)
    bsop_neon_intrin(&answer_fptr[i], &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &r_fptr
    [i], &sigma_fptr[i], &T_fptr[i]);
    #else
    answer_fptr[i] = bsop_reference_float(cpflag_fptr[i], S0_fptr[i], K_fptr[i], r_fptr
    [i], sigma_fptr[i], T_fptr[i]);
    #endif
}
```

stead, instructions for estimating the reciprocal of a number, as well as the reciprocal of the square root is available. The reciprocal of the elements in some vector d is found by first obtaining a rough estimate using the intrinsic `r = vrecpeq_f32(d)` [65], and then applying the Newton-Raphson method by iterating $x_{n+1} = x_n(2 - x_n d)$ [66]. The function $2 - x_n d$ is implemented as the reciprocal step instruction `vrecps`, with the intrinsic `vrecpsq_f32(r, d)`, and thus the value of the reciprocal `r` can be improved by applying `r = vmul_f32(r, vrecpsq_f32(r,d))`.

For computing the square root in NEON, the process is similar to that of the reciprocal. An initial estimate for the reciprocal of the square root of some floating point vector d is found using the intrinsic `s = vrsqrteq_f32(n);` [65]. Then the Newton-Raphson iteration $x_{n+1} = x_n(3 - dx_n^2)/2$ is applied until a sufficiently accurate estimate of the reciprocal of the square root is obtained [66]. Finally, the reciprocal of the value obtained is computed, and the result from this is the square root. The Newton-Raphson iteration can be performed using the `vrsqrtsq_f32(s, d)` intrinsic, which computes the function $3 - dx_n/2$. The full iteration using intrinsics is then `vmulq_f32(s, vrsqrtsq_f32(s, vmulq_f32(d,d)));`.

In the NEON implementation of Black-Scholes in this thesis, the Newton-Raphson iteration is performed four times per reciprocal, and four times per reciprocal square root. Since Newton-Raphson converges quadratically, the estimated number of correct digits is 16 with a good first guess, which is provided by the intrinsics `vrecpeq_f32` and `vrsqrteq_f32`.

Negation and absolute value operations are not a part of the SSE or AVX instruction set. The IEEE 754 standard for floating point numbers specify that the first bit is the sign bit [67]. A vector negation operation for floating point can therefore be formed by simply XOR-ing each element by the mask `0x80000000` (in binary, `0b100...000`); this is equivalent to flipping the first bit of each element and leaving the rest of the bits alone, as $1 \oplus x = \neg x$, and $0 \oplus x = x$.

An absolute value operation can be implemented in a similar fashion as negation, but instead of bitwise XOR, use bitwise AND, and modify the mask to `0x7fffffff`, or in binary, `0b011...111`. Effectively, this clears the sign bit, because $0 \wedge x = 0$, and $1 \wedge x = x$.

Note that these operations would not work for integers as negative integers are encoded as 2's complement, and thus the rest of the bits would be different from the positive number. The SSE and AVX code for the absolute value and negation operations for floating point vectors in SSE and AVX is listed in listing 4.6. For NEON, instructions exist for taking the absolute value and for negation.

Listing 4.6 Absolute value function and negation on SSE/AVX

```
/* SSE */
//Set the mask to 0111...11 on every element, and cast it to a float
__m128 absmask = _mm_castsi128_ps(_mm_set1_epi32(0x7fffffff));
x = _mm_and_ps(absmask, x);

/* AVX */
__m256 _absmask = _mm256_castsi256_ps(_mm256_set1_epi32(0x7fffffff));
x = _mm256_and_ps(absmask, x);
```

(a) Vector implementation of absolute value

```
/* SSE */
//Set the mask to 1000...00 on every element, and cast it to a float
__m128 negmask = _mm_castsi128_ps(_mm_set1_epi32(0x80000000));
x = _mm_xor_ps(absmask, x);

/* AVX */
__m256 negmask = _mm256_castsi256_ps(_mm256_set1_epi32(0x80000000));
x = _mm256_xor_ps(negmask, x);
```

(b) Vector implementation of negation

4.4 Fastest Fourier Transform in the West

In order to do benchmarks with FFTW using OmpSs, the FFTW library had to be modified to use OmpSs, and a benchmark had to be written to call the library functions. SSE and AVX is already in place in FFTW, so the only change necessary was going from using OpenMP to OmpSs.

4.4.1 Modifications to the FFTW library

The Mercurium compiler, `sscc`, does not support the data type `__float128` which is used in the FFTW3 header to define the quad precision complex data type resulting in a compilation error. Since quad precision is not used in this project, the line using the `__float128` data type was simply commented out.

To make the benchmark also use tasks and to remove constructs obsoleted by OmpSs like the `#pragma omp parallel` blocks, the OpenMP code for the benchmark in the file `threads/openmp.c` in the FFTW source tree was modified. First, the `#pragma omp parallel for private(d)` directive was removed in the `spawn_loops` function. Then, a new `#pragma omp task private(d)` directive was put inside the

Argument	Description
--width=N	Set the input width to N
--inplace	Call the FFTW kernel in-place (i.e. the output is placed in the input array)
--measure	Make FFTW plans with auto-tuning to the system. This involves running a number of transforms to explore different plans.

Table 4.2: FFT benchmark arguments

for loop, in order to spawn a new task for each iteration. Then finally a `#pragma omp taskwait` directive was inserted after the for-loop. The final loop is shown in listing F.1.

4.4.2 Implementation of the benchmark

As FFTW is a library an external benchmark application is used that calls FFTW. A FFTW benchmark exists called benchFFT, which runs FFTW and several other FFT libraries for comparison of performance [68]. However, benchFFT is designed to automatically run many standardized tests on many different FFT implementations, which does not fit the need of this project.

Instead, a custom benchmark that suits the needs of this project was developed. All this benchmark does is take the input size as a command line argument along with modifiers, and then create a FFTW plan for execution, and finally execute the plan on the problem given. The contents of the input array are populated with random complex numbers between $-10000 - 10000i$ and $10000 + 10000i$ before execution starts.

The benchmark takes in a number of arguments, with the most important ones shown in table 4.2. If run with --measure, the first execution will take longer because FFTW will explore different combinations of parameters. However, after a problem size has been planned out once for a set number of threads and SIMDization settings, the benchmark application will store the knowledge that FFTW gained to disk, called wisdom in the FFTW terminology, so that subsequent benchmarks with the same parameters will incur only a minor overhead comparable to running without --measure.

Listing F.2 shows a condensed version of the source code for the FFTW benchmark. One important thing to note is that when operating on input sizes smaller than 8192, the benchmark still allocates space for 8192 elements. This is because it was found that with smaller allocations, the performance for these small problem sizes was very low (less than 0.2 GFLOPS for 2048 and 4096 elements), causing a sudden increase in performance when going from 4096 to 8192 elements, from about 0.2 GFLOPS to about 3. Always allocating 8192 or more elements solved this problem. We attribute this to destructive cache interference for such small problems.

4.5 Matrix Multiplication

The matrix multiplication application was developed at the Barcelona Supercomputing Center (BSC) as a prototype application that uses OmpSs, and this application was made available. Code for energy measurements and for reading cache miss counters was added. The application has two main parameters: The block size that each task will compute, and the problem size. This block size is partly independent from the block sizes used to perform cache-efficient matrix-matrix multiplications, as the blocks are merely there to provide parallelism. ATLAS performes cache-efficient blocking of each block-block multiplication.

The algorithm does the following. First, the matrices A, B and C are subdivided into $N/NB \times N/NB$ blocks. The multiplication of any two blocks is defined as one task, and any two multiplications that are not stored in the same block of C can be performed in parallel. Then, a blocked matrix-matrix multiplication is performed, where each block-block multiplication is performed by a call to ATLAS. ATLAS in turn may subdivide each block in order to make them fit in the L1 or L2 cache. The pseudo-code for this algorithm is shown in algorithm 4. `sgemm` refers to the BLAS procedure "Single precision GEneral Matrix-Matrix multiply", and is the call to ATLAS.

Algorithm 4 Task-based Matrix-Matrix Multiply

```
procedure MATMUL($A$, $B$, $C$, $N$, $b$)
    for $i := 0 \to N/b - 1$ do
        for $j := 0 \to N/b - 1$ do
            for $k := 0 \to N/b - 1$ do
                $ia \leftarrow i \cdot b$
                $jj \leftarrow j \cdot b$
                $kk \leftarrow k \cdot b$
                Create task SGEMM($A[ii : ii + b, kk : kk + b]$, $B[kk : kk + b, jj : jj + b]$,
$C[ii : ii + b, jj : jj + b]$, $b$)
            end for
        end for
    end for
end procedure
```

Chapter 5

Experiment Setup and Methodology

This chapter describes the experiment setup for the benchmarks. In addition to giving an overview of the hardware and software packages used, the methodology of experiments, FLOPS counts and statistical metrics are covered.

5.1 Test Bench

Experiments were run on `sif`, a workstation with a quad core Sandy Bridge CPU. Additionally, experiments were run on an ARM Motherboard Express μATX development board equipped with a CoreTile Express A9x4 daughter board with a Cortex-A9 MPCore test chip, a quad-core CPU. The following subsections have an overview of the hardware for these machines, which compilers was used and compiler flags and which third-party libraries and software that were used.

5.1.1 Hardware

sif

`sif` has a Intel Sandy Bridge Core i7 2600 CPU with four cores, but with the option of turning on hyper-threading for an additional four hardware threads. A diagram of the cores and caches are shown in figure 5.1. The hardware specifications for `sif` are listed in table 5.1. The CPU specifications were mostly retrieved from `/proc/cpuinfo`. Cache sizes, line sizes and ways of associativity were retrieved from `/sys/devices/system/cpu/cpu0/cache/index*`. Cache latencies were found in [69, table 2.6].

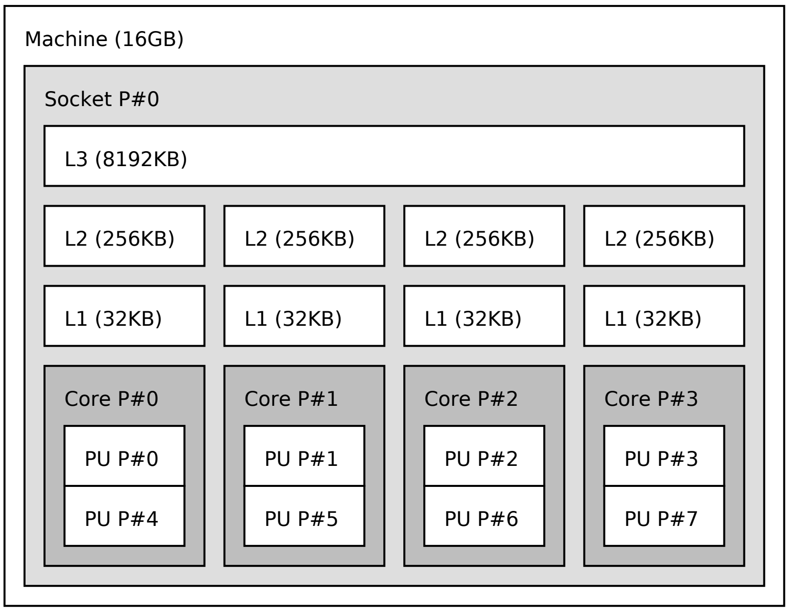


Figure 5.1: Diagram of caches, cores and hyper-threads on sif

Property	**Value**
CPU model	Intel(R) Core(TM) i7-2600
Model #	42
Stepping	7
Manufacturing process	32nm
Clock frequency (min-max)	1.60GHz - 3.40GHz
Number of physical cores	4
Number of logical cores	8
Main memory	16GB

Table 5.1: Hardware specifications for sif

Cache	**Size**	**Ways of associativity**	**Line size**	**Latency (cycles)**
Level 1	32KB(Data)/32KB(Instr)	8	64B	4
Level 2	256KB	8	64B	12
Level 3	8MB (shared)	16	64B	26-31

Table 5.2: Cache information for Intel Core i7-2600, 3.4GHz

ARM development board

The ARM development board has a daughterboard with four Cortex-A9 CPU cores. The hardware specifications are listed in table 5.3, and a diagram of the hardware is provided in figure 5.2.

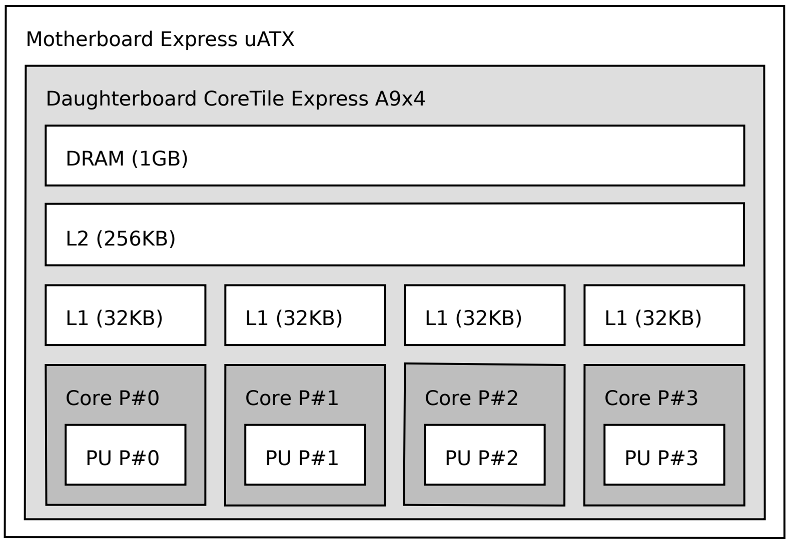


Figure 5.2: Diagram of caches, cores and memory on the ARM board

Property	**Value**
Baseboard	ARM Motherboard Express μATX
Daughter board	CoreTile Express A9x4
Main memory	1GB
CPU model	ARM Cortex-A9 MPCore
Clock frequency (max)	400MHz ([70])
Number of physical cores	4
Number of logical cores	4
Level 1 cache (Data/instr)	32kB/32kB [71]

Table 5.3: Hardware specifications for ARM development board

5.1.2 Software and Libraries

On sif, openSUSE 11.4 (x86_64) is used with Linux kernel version 2.6.37.6. On the ARM board, Debian Wheezy with kernel 3.3.0 was used. Also, a list of third party software packages that were used in this thesis is listed in table 5.4.

Software	**Version**	**Licence**	**Website**
Nanos++	0.6a/0.7a	GNU LGPL	https://pm.bsc.es/projects/nanox
Mercurium	1.3.5.8	GNU LGPL	https://pm.bsc.es/projects/mcxx
FFTW	3.3.1-beta1	GNU GPL[1]	http://www.fftw.org
ATLAS	3.9.70	BSD-style	http://math-atlas.sourceforge.net
sse_mathfun	Unknown	zlib Licence	http://gruntthepeon.free.fr/ssemath
PAPI	4.2.1	BSD Licence	http://icl.cs.utk.edu/papi

Table 5.4: Third party software and libraries

[1]GNU General Public Licence

5.1.3 Compiler and compiler flags

For compilation of OmpSs code, the Mercurium compiler, `sscc` version 1.3.5.8 was used. `sscc` in turn calls `gcc` for compilation of the translated code to native machine code. `gcc` version 4.6 was the first version of `gcc` to get support for AVX intrinsics and data types. `gcc` version 4.7 was used on the Sandy Bridge machine. On the ARM development board, GCC 4.6.3 was used. Compiler and configure flags for the codes compiled in this thesis are listed in table 5.5 and 5.6, respectively. For compilation of the software that comprises OmpSs, i.e. the Mercurium compiler and the Nanos++ runtime system, `gcc` was used. Note that the software's `configure` script may add additional flags automatically that are not listed here.

Software	**Compiler suite**	**Compiler flags**	**Remarks**
Common flags for all software		ARM: `-mcpu=cortex-a9 -mfpu=neon` Intel: `-march=corei7-avx`	
Black-Scholes	`sscc`	`-O3 -std=c99 --ompss`[1]	See footnote.
FFTW (benchmark)	`sscc`	`-O3 -std=c99`[2]	See footnote.
Matrix multiplication	`sscc`	`-O3 -std=c99 --ompss`	
OmpSs Package			
Bison	`gcc`	`-O2`	
Nanos++ runtime	`gcc`	`-O2`	
Mercurium	`gcc`	`-O2`	
Other Packages			
FFTW	`sscc`	`-O2 --ompss`	
ATLAS	`gcc`		
PAPI	`gcc`		

Table 5.5: Compiler flags for software used in the thesis

Software	**./configure flags (excl. paths)**

Mercurium	`--enable-tl-openmp-nanox`
	`--enable-ompss --enable-tl-superscalar`
	`--with-superscalar-runtime-api-version=5`

FFTW	`--enable-openmp {--enable-avx|--enable-sse2|--enable-neon}`
ATLAS	None

Table 5.6: Configure flags for third party software

[1]Addition for AVX/SSE: `-mavx -DAVX`/`-msse2 -DSSE`
[2]Addition for AVX/SSE: `-DAVX`/`-DSSE`

5.2 Experiment Methodology

Before any benchmarking was begun, the kernels were run once after initialization in order to avoid cold start cache misses. Next, the performance counters for cache statistics are started with `PAPI_start_counters()` and energy measurements are started with a call to `start_reading()` and stopped with `stop_reading()`, which is the functions that implement the algorithms described in section 4.1. Then, the current system time is retrieved using the POSIX function `gettimeofday()`.

If a kernel runs for less than 0.1 seconds, it is repeatedly run until 0.1 seconds has passed. The values for running time, cache misses/references and energy usage were divided by the number of runs to get the average. This was done in order to get accurate readings from the energy registers, which has a limited resolution of $15.3\mu J$ [56, Vol 3B, sec. 14.7.1 RAPL Interfaces] as default, as well as a limited update frequency.

Each application was run 10 times for each configuration of problem size, number of threads, optimizations and other flags. The median of the measurements was used in the results that are presented.

Unless otherwise specified, the CPU was run at full clock frequency (3.4GHz for the Sandy Bridge, 400MHz for the ARM Cortex-A9 MPCore). To ensure that there would be no interruption from automatic frequency scaling, the frequency was fixed to 3.4GHz before each run when using the Sandy Bridge. No frequency scaling was enabled on the Cortex-A9.

5.2.1 Energy Measurements

For Sandy Bridge, when an experiment is run, the current state of the `MSR_{PKG|PP0|PP1}_-ENERGY_STATUS` register is read. After the experiment ends, the same register is read again. The difference between these two readouts is the energy spent during the experiment, divided by the energy status unit, as described above. To get the energy spent in Joules, the value from the register is multiplied by the energy unit as described in section 4.1. To get the average power, the energy in Joules is divided by the duration of the experiment.

For ARM, before the experiment starts, a energy sampling thread is spawned which implements algorithm 2 from section 4.1 and is run continuously until the benchmark ends. For each sampling iteration, both device 0 and 1 (L2 cache+SRAM, and CPU cores) is measured, using algorithm 1. After the benchmark ends, the package energy is computed as $E_{pkg} = E_{cores} + E_{L2+SRAM}$.

Every experiment run for performance measurements also records the energy usage in Joules.

5.2.2 Experiments

In Black-Scholes, the task size, i.e. the number of elements of the output array each task computes, is varied for different scheduling algorithms for the small problem size using eight threads for Sandy Bridge, and four threads on ARM. In FFTW, only scheduling algorithms are tested as there are no task size parameter. Scheduling algorithms are chosen by setting the `NX_SCHEDULE` environment variable. Task sizes are set by setting the `--lwgsize` parameter to the Black-Scholes application.

In Black-Scholes and FFTW, two specific problem sizes were chosen: One small ($N = 2^{13}$ for Black-Scholes, and $N = 2^{14}$ for FFTW) and one large ($N = 2^{25}$ for both on Sandy Bridge, and $N = 2^{24}$ for both on ARM). The purpose of the large problem sizes is to see how well the application performs when the problem is much larger than what fits in the cache. The small problem is small enough to fit in the L2 cache, and thus should see better speedup since the bottleneck with memory bandwidth and latency is reduced. These problem sizes were run on one up to eight threads on Sandy Bridge, and four threads on ARM. For ARM, smaller problem sizes are used due to both memory limitations, and due to more limited processing power.

In addition to testing two specific problem sizes on every thread configuration, a range of problem sizes from $N = 2$ up to $N = 2^{25}$ was used on both FFTW and Black-Scholes with one, four and eight threads on Sandy Bridge. For ARM, $N = 2$ and up to $N = 2^{23}$ was tested with one, two and four threads. The reason for this benchmark is seeing the performance varying as the problem size grows to see the effects of the cache, and how this relates to cache misses. For matrix multiplication, the matrix sizes were chosen to be 128×128, 256×256, 512×512, ..., 8192×8192 for Sandy Bridge, and 64×64 up to 2048×2048 on ARM. Larger problem sizes takes too long to compute to be practical for benchmarking on a single node as the number of floating point operations are $O(N^3)$.

Nanos++ version 0.7a were used for all results except those in section 6.1. Version 0.6a and 0.7a show very different performance results on small problem sizes on both Black-Scholes and FFTW. As such, the experiments using the small problem size is run on 0.6a as well as 0.7a.

All tests were performed in single (32 bit) precision because the instruction set in the ARM Cortex-A9 MPCore, ARMv7, does not support double precision with NEON vector instructions. However, the ARMv8+A instruction set will support double precision floating point [72].

Stability of the energy MSRs on Sandy Bridge

In order to get trustworthy results for energy measurements, the MSRs must deliver stable results; i.e. the standard deviation should be small. To establish that the MSRs are stable, a benchmark computing the matrix product $C = AB$ where A and B is 512×512 matrices were run for 1000 iterations, and the standard deviation was computed. ATLAS was used to perform the multiplication. The distribution of the energy measurements are shown in figure 5.3 and the statistics are tabulated in table 5.7. The data show that generally the measurements are very stable, with a standard deviation of less than 1% of the mean.

An outlier in these figures is defined as a value with a difference from the mean larger than three standard deviations. Assuming a normal distribution of the measurements, approximately $1/370$ of the measurements are expected to be outliers with this definition due to the three-sigma rule which states that $Pr(\mu - 3\sigma \leq x \leq \mu + 3\sigma) \approx 0.9973$. Two outliers are detected, which is within the expected range for a normal distribution.

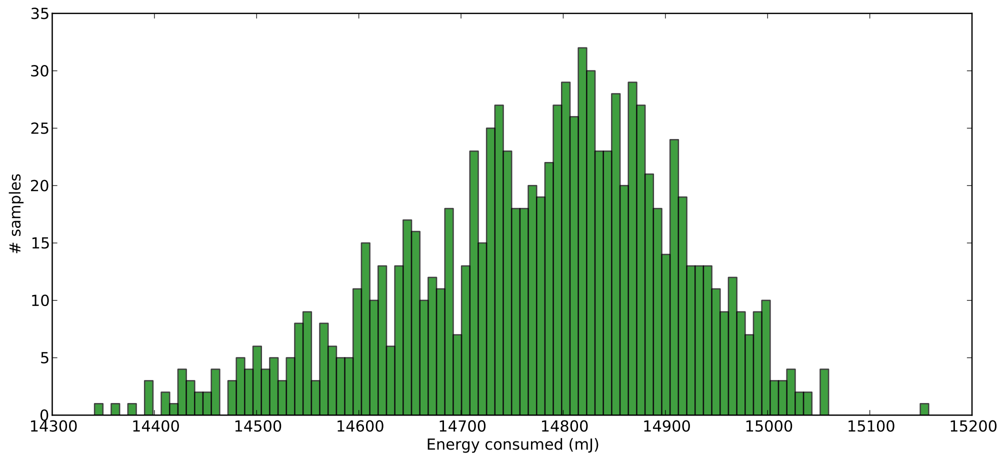


Figure 5.3: Distribution of sample values from MSR consistency test

Median	Mean	Standard deviation	Minimum	Maximum	Outliers
14795.59	14774.94	135.35	14341.41	15157.57	2

Table 5.7: Statistical data from MSR consistency test

Stability of sampling method on ARM

The stability of the energy measurements were also measured on the ARM platform, using power sampling as described in section 4.1. The same benchmark as for Sandy Bridge was used, but instead of multiplying 512×512 matrices, 128×128 matrices were used. The results are shown in figure 5.4, and a statistical summary is given in table 5.8. As with for Sandy Bridge, the measurements are stable, with two outliers and a standard deviation of less than 0.8% of the mean.

Median	Mean	Standard deviation	Minimum	Maximum	Outliers
119.89	119.92	0.95	117.74	132.27	2

Table 5.8: Statistical data from consistency test of energy measurements on ARM

Discussion of wrap-around times of MSRs

The Intel architectures software developer manual gives an estimate of the wrap-around time for the package energy status register of 60 seconds on high load [56, Vol 3B, sec. 14.7.3 Package RAPL Domains]. However, calculations indicate that the wrap-around

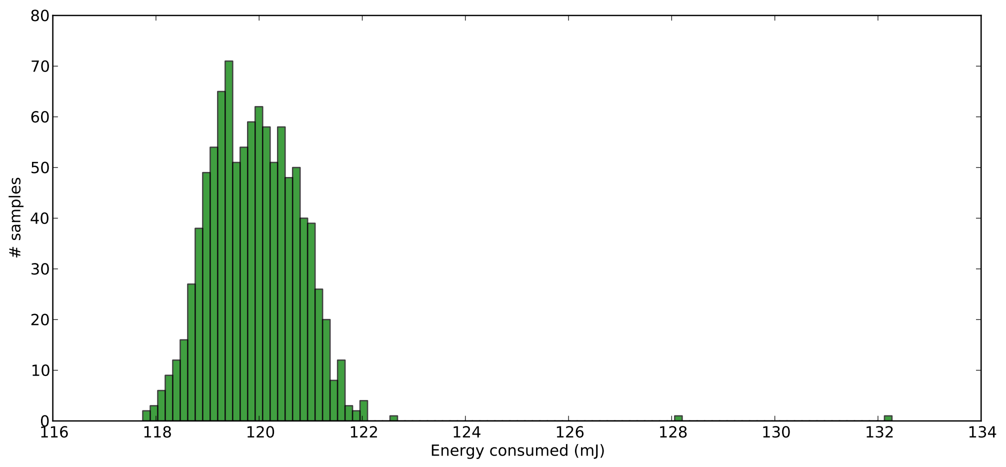


Figure 5.4: Distribution of sample values from energy measurements using sampling on ARM

time may be far greater. First, assuming default granularity of $15.3\mu J$ on the energy status registers, these registers may represent values between $0J$ and $15.3 \cdot 1000^{-2} \cdot (2^{32} - 1)J \approx 65713J$. Assuming that the CPU uses the maximum thermal design power (TDP) of the benchmarked CPU, $95W$ [73], we can find the wrap-around time to be $65713J/95W \approx 691.7s$, more than a factor of 10 larger than the estimate in the manual.

5.2.3 Cache behavior experiments

The Intel Sandy Bridge CPU has performance counters available for measuring many performance-related statistics like the number of instructions executed, the number of FLOPS performed, cache misses/cache references, and more [56]. Performance counters are used in this work to measure the cache miss and reference rate for the L2 and L3 cache.

Performance counters counts events of certain types, e.g. cache misses, cache accesses, load/stores, and so on. PAPI (Performance Application Programming Interface) [74] is a high-level library for reading the counters counting occurrences of these events. It provides a set of predefined events; e.g. `PAPI_L2_TCA` is all level 2 cache accesses, and `PAPI_L2_TCM` is all level 2 cache misses. Table 5.9 lists the events used for measuring level 2 and 3 cache miss rate. The native events were retrieved by running the program `papi_decode`.

PAPI event	**Native event** [56, table 19-3]	**Description**
`PAPI_L2_TCA`	`L2_RQSTS:ALL_CODE_RD` + `L1D:REPLACEMENT`	Total L2 cache accesses
`PAPI_L2_TCM`	`LONGEST_LAT_CACHE:REFERENCE`	Total L2 cache misses
`PAPI_L3_TCA`	`LONGEST_LAT_CACHE:REFERENCE`	Total L3 cache accesses
`PAPI_L3_TCM`	`LONGEST_LAT_CACHE:MISS`	Total L3 cache misses

Table 5.9: Performance counter metric composition of PAPI events

From table 5.9 we see that the number of references to the L2 cache is implemented by adding the number of replacements in L1's data cache (i.e. the number of times lines were replaced in L1D due to misses) and the number of instruction reads from L2. L2 miss count is implemented as the number of references to the longest latency cache (i.e. the L3 cache); this is a natural choice since the L3 is accessed when and only when there is a miss in L2. The L3 total accesses and L3 total misses are self-explanatory from the table.

The Cortex-A9 does not offer performance counters to compute the cache miss rates other than the number of L1 misses.

5.3 Defining FLOPS Counts

In order to measure the computational throughput in FLOPS it must be defined how many floating point operations goes into computing the answer using a given input size. One way of obtaining this would be to use performance counters to measure how many floating point operations was actually performed for each experiment. However, this also counts unnecessary operations due to non-optimal algorithms, and is not well suited for comparing performance between implementations. For instance, consider a hypothetical case where some algorithm has two implementations, A and B. Say B runs 10% slower in terms of running time compared to A, but performs 20% more floating point operations. Implementation A is clearly better since it completes faster, but B would have a higher FLOPS rate, which gives a false view of the performance of the algorithm.

Another way is counting the number of floating point operations in the algorithm (or an idealized version of the algorithm), by simply counting add, multiply, divide and subtract-operations. This way, the number of FLOPS an application manages to achieve is directly proportional with the running time, and thus is a good measure of how much useful work an algorithm actually does. One example to illustrate the difference between useful and actual floating point operations is in parallelizing stencil methods [75]. When parallelizing stencil methods, border cells for each processor (ghost cells) needs to be communicated to each neighboring processor. When data movement is expensive due to high latencies, it may pay off to transfer more than one layer of ghost cells to the neighboring processors so that fewer data transfers needs to take place. Then each processor needs to do extra work because some cells then is computed simultaneously by more than one processor, and thus the actual work is larger than the amount of useful work.

For this thesis, FLOPS measurements were done by counting or estimating the number of "useful" floating point operations, and dividing this by the running time. Integer operations such as shifts and bitwise logical operations, and comparisons were ignored. Table 5.10 shows the floating point operation counts for each application. For Black-Scholes, assumptions were made for how many floating point operations the functions `log`, `exp` and `sqrt` do. These numbers were mainly retrieved from [76], but were also checked against the Cephes math library [61], and the numbers were confirmed to be underestimates of the actual floating-point operation counts for these functions by visual inspection of the code. For completeness, these are also tabulated in

5.10.

Kernel	**Problem size**	**Floating-point operation count**	**Remarks**
exp	1	20	From [76].
log	1	20	From [76].
sqrt	1	15	From [76].
Black-Scholes	N	$153N$	From counting, using the above assumptions about square root, log and exp.
FFTW	N	$5N \log N$	From [17].
Matrix multiplication	N	$2N^3$	From equation 3.6.

Table 5.10: Floating-point operation counts for different kernels/functions

5.4 Problem Sizes and Memory Footprints

In each experiment, the problem sizes are represented by the value N. What N says about the computational complexity and space requirements varies between applications. In the Black-Scholes benchmark, there are six input arrays and one output array where each element is four bytes and N is here the number of elements in each array. This gives a memory footprint of Black-Scholes of $4 \cdot 7 \cdot N = 28N$ bytes. In FFTW N represents the number of elements in the array that are to be transformed to the Fourier domain. The benchmark uses an out-of-place transform, requiring one output array and one input array, each containing N elements of $4 \times 2 = 8$ bytes each, representing each single-precision complex number. This gives a total memory footprint of $16N$. For matrix multiplication, N represents the number of elements in either dimension for the input and output matrices A, B and C. Three matrices of size $N \times N$ must be stored, requiring a total of $4 \cdot 3N^2 = 12N^2$ bytes.

5.5 Statistical Metrics

Each experiment is run for 0.1 seconds, and depending on variability, each iteration may run the kernel a different number of times. Because of this, weighted means and weighted standard deviations is used to compute the mean and standard deviation, respectively.

The weights are chosen to be $w_i = N_i$, where N_i is the number of times the kernel ran in iteration i. Let k be the number of iterations, and $N = \sum_i N_i$. This gives an estimator of the mean $\hat{\mu}$ as

$$\hat{\mu} = \frac{1}{N} \sum_i N_i X_i$$

and an estimate of the standard deviation of the mean $\hat{\sigma}$ as

$$\hat{\sigma}_{mean} = \sqrt{\frac{k}{N(k-1)} \sum_i N_i (X_i - \hat{\mu})^2}$$

It is easy to see that if each iteration only runs once, i.e. $N_i = 1$, then $N = k$, and this formula reduces to the commonly used formula for the sample standard deviation,

$$\hat{\sigma} = \sqrt{\frac{1}{N-1}\sum_i (X_i - \hat{\mu})^2}$$

In this report, the relative standard deviation, in percent (%RSD) is reported for the experiments. This quantity is defined as

$$\%RSD = 100\% \cdot \frac{\sigma_{mean}}{\hat{\mu}}$$

Chapter 6

Results and Discussion

In this chapter we present performance results with discussion for Sandy Bridge. ARM results can be found in appendix A.

Performance was tested for all kernels, both with regards to vectorization with SSE and AVX, and multi-threading. Performance testing is necessary for energy efficiency studies, as energy efficiency often is considered a function of performance and energy usage. For instance, the energy-delay product involves the total running time as well as energy consumed, and the $FLOPS/Power$ metric is a function of power (J/s) and performance (FLOPS).

Performance can be improved in several ways: Adding more cores, vectorizing the code using SIMD instructions, increasing the clock frequency, optimizing the existing code and using a more efficient compiler. However, decreasing the divisor, i.e. power is more difficult because that often involves e.g. decreasing the clock frequency, which impacts performance. Instead, we may try to increase efficiency through better utilization of CPU functionality like vectorization, and using multiple cores instead of increasing the clock frequency.

One of the arguments for using multiple CPU cores is that single core performance at some point will reach an ILP wall[1], and increased power requirements due to higher clock frequencies [30][32][31]. Assuming perfect parallelism within the application, the theoretical speedup is N using N CPU cores. However, Amdahl's Law states that speedup is limited by the serial fraction of the program [77]. There are often dependencies between the tasks, and two dependent tasks must be executed serially. Performance is also often limited by memory bandwidth.

The following sections presents performance and energy efficiency results of Black-Scholes, FFTW and matrix multiplication for Sandy Bridge. Results are presented both with respect to the number of cores, and vectorization using SIMD instructions. First, performance results using an older version of Nanos++, 0.6a, is presented. Then, the results for the three applications Black-Scholes, FFTW and matrix multiplication are presented.

[1]ILP = Instruction level parallelism

For each application, the results are presented in the order given by table 6.1. Details about the experiments is found in section 5.2.2.

	Black-Scholes and FFTW
1.	Performance using different scheduling algorithms, and on Black-Scholes, different task sizes
2.	Performance for one through eight threads
3.	Performance as problem size increases
4.	Power dissipation for one through eight threads
5.	Power dissipation for a range of problem sizes
6.	Energy efficiency in GFLOPS/W for a range of problem sizes
7.	Energy consumed for one through eight threads
8.	Normalized EDP and EDD for one through eight threads
	Matrix multiplication
1.	Performance as problem size increases
2.	Power dissipation for a range of problem sizes
3.	Energy efficiency in GFLOPS/W for a range of problem sizes
4.	Normalized energy consumption for a range of problem sizes
5.	Normalized EDP and EDD for a range of problem sizes

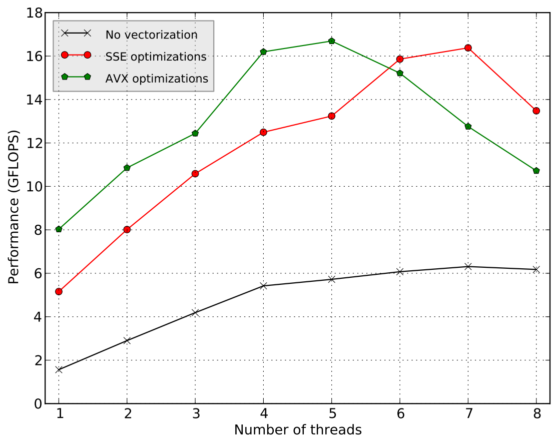
Table 6.1: Result presentation order

6.1 Black-Scholes: Scheduling and Variability in Nanos++ Version 0.6a

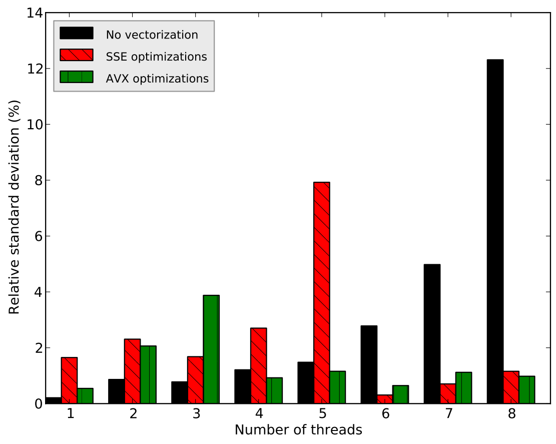
In Nanos++ version 0.6a, significant variation in the performance was observed for the default scheduling algorithm. This section presents the results from the experiments that was performed using 0.6a. Although the newer version does not show this behavior, the newer version show significantly lower performance for small problem sizes compared to the old Nanos++ version when the scheduling algorithms and parameters have been tweaked for both.

Figure 6.1 and 6.2 shows the performance and standard deviation of the performance of Black-Scholes evaluations when going from 1 to 8 threads. For the large problem in figure 6.1, the performance is mostly stable, although it does drop after five threads. However, for the small problem, we see that we get some speedup going from one to two threads, but after this, the performance drops except for the non-vectorized version, before going back up at five and seven threads. Looking at the standard deviation, the values are very high, with up to almost 80% relative to the mean. Note that as mentioned in section 5.5, each experiment is run for at least 0.1 seconds, and the running times are then averaged, so the standard deviation of each separate run is even higher than what is shown here.

We attribute the low performance and high variability of the small problem is that the default scheduling policy used in OmpSs has a weakness when an application is run for a sufficiently small amount of time. Two other scheduling policies was tested

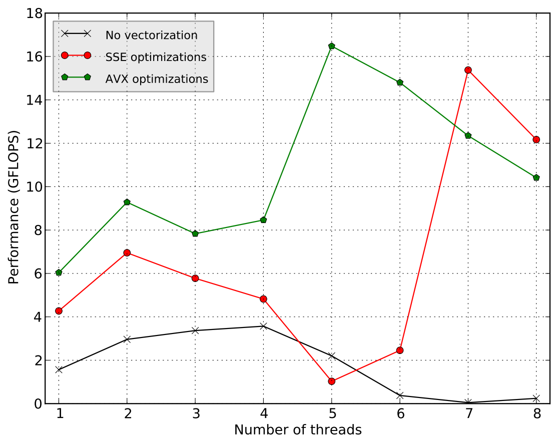


(a) Performance

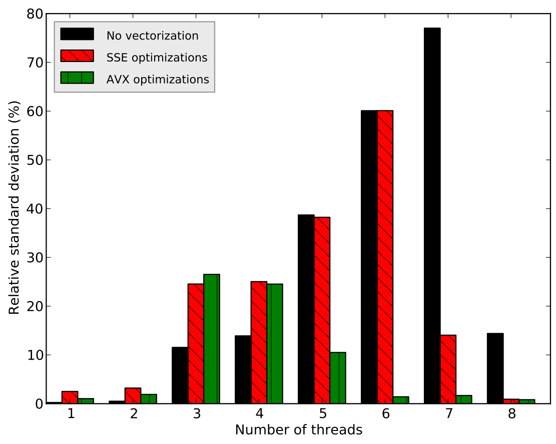


(b) Standard deviation

Figure 6.1: Performance with regards to number of threads, Black-Scholes, N=2^{25}



(a) Performance



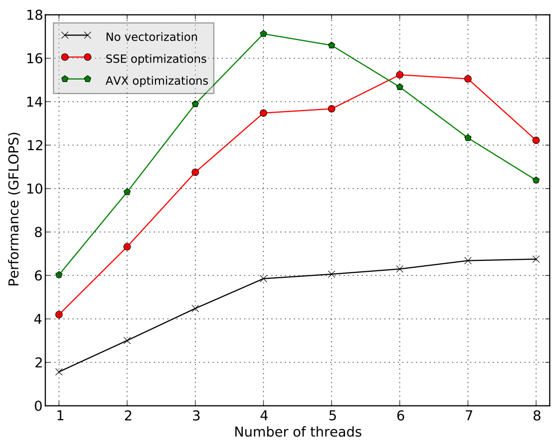
(b) Standard deviation

Figure 6.2: Performance with regards to number of threads, Black-Scholes, N=2^{13}

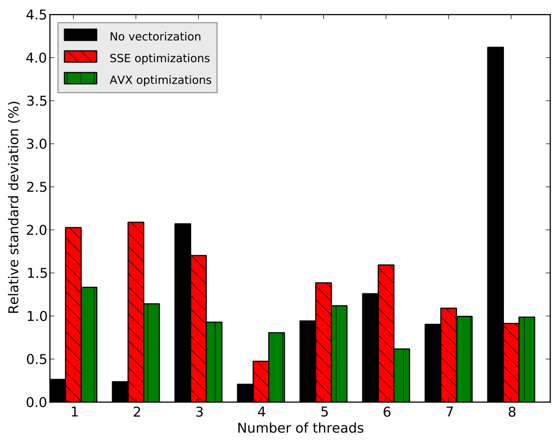
using the small problem size: Distributed breadth-first, and work-first, both described in section 2.2.5. The results are shown in figures 6.3 and 6.4.

We see that the standard deviation is clearly smaller than using the default scheduling, with distributed breadth-first scheduling having considerably smaller standard deviations on most configurations than work-first. When running on two threads, work-first shows a considerable degradation of performance, accompanied by a high standard deviation. Overall, the performance with work-first scheduling is lower than distributed breath-first.

What both work-first and distributed breadth-first schedulers have in common is that they implement work stealing for idle threads, which results in less time where threads actually idle. When a thread is idling either because there are no ready tasks and there is no task stealing, the scheduler may do one of two things: Actively wait for work by spinning until work is available, or yielding to the operating system's thread

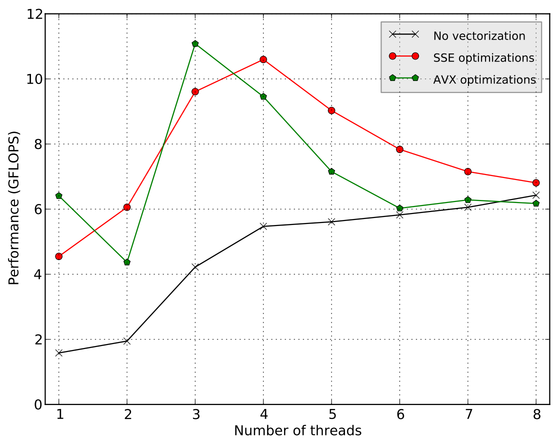


(a) Performance

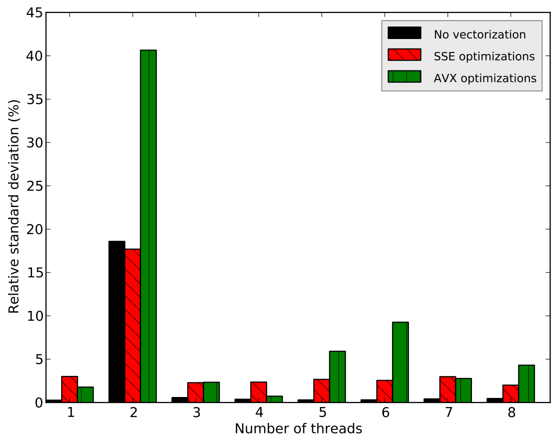


(b) Standard deviation

Figure 6.3: Performance of small problem, distributed breadth-first scheduling



(a) Performance

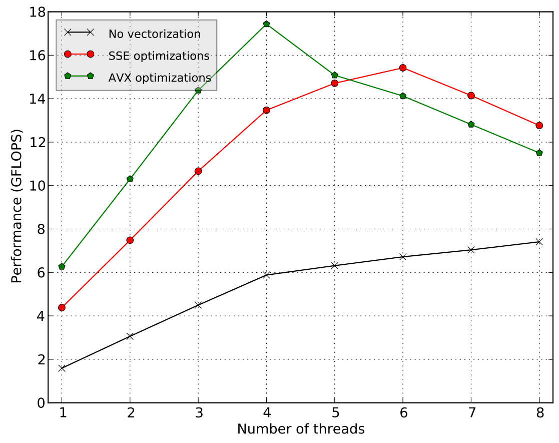


(b) Standard deviation

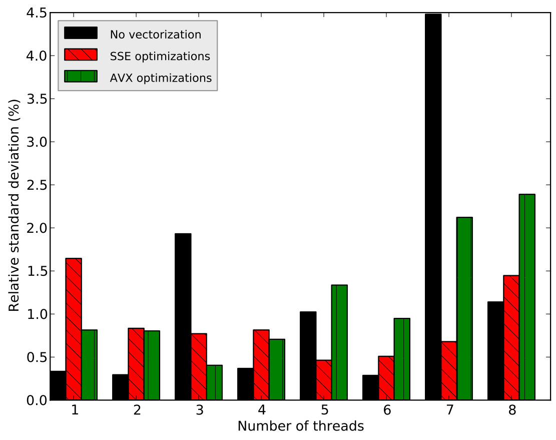
Figure 6.4: Performance of small problem, work-first scheduling

scheduler until there is available work. The latter gives a slight overhead because if work becomes available shortly after the thread yielded, the thread will not do any work before the operating system's thread scheduler context switches that thread back in again. On the other hand, spinning may give some overhead when sharing resources on a single core like when a processor runs with hyper-threading enabled. When no hyper-threading is active, we have no explanation for the behavior we see.

As a compromise between always yielding and always spinning, the Nanos++ scheduler spins for a certain number of times, before yielding to the operating system. Some investigation into the Nanos++ runtime system reveals that when idle, the scheduler spins 100 times before yielding; this value is controllable by the environment variable `NX_SPINS`. In order to remove the factor of resource contention on hyper-threads due to the scheduler spinning, `NX_SPINS` was set to 1. Figures 6.5, 6.6 and 6.7 shows these results.

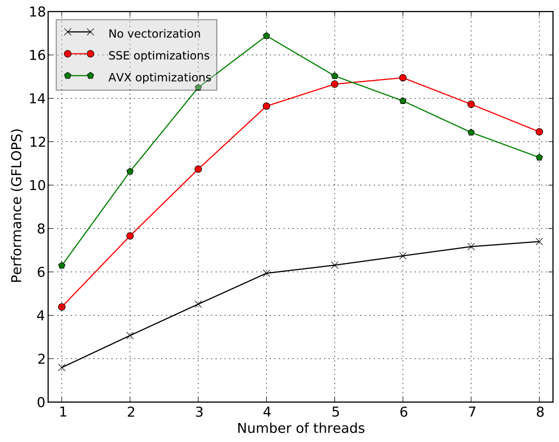


(a) Performance

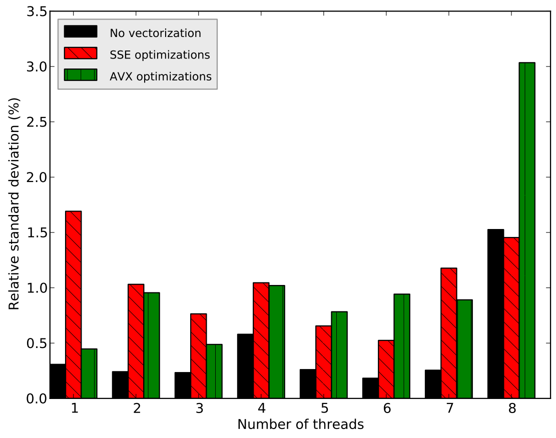


(b) Standard deviation

Figure 6.5: Performance of small problem, default scheduling, no spinning



(a) Performance

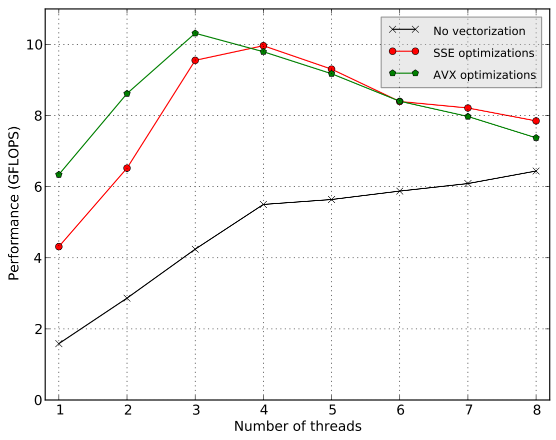


(b) Standard deviation

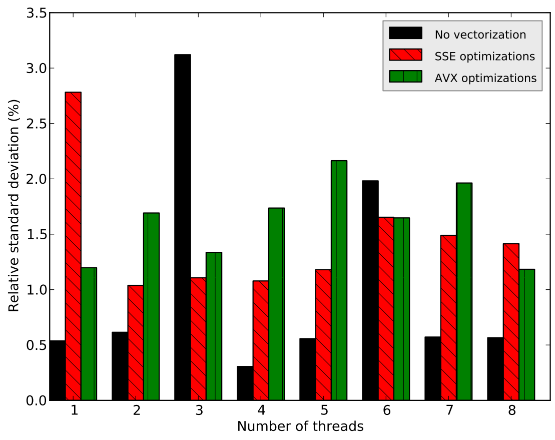
Figure 6.6: Performance of small problem, distributed breadth-first scheduling, no spinning

The results clearly show far more stable results than keeping `NX_SPINS` at the default value. Work-first scheduling again show poor performance compared to distributed breadth-first, while the default and DBF schedulers give similar results performance-wise, and both show an overall low standard deviation. The default scheduler shows a higher performance than the DBF scheduler, but only slightly. We now see a definite speedup going up to four threads for all vectorizations, however only the SSE and non-vectorized implementation continues to speed up after four threads.

The results using different task sizes on different scheduling algorithms in Black-Scholes are shown in figure 6.8. The performance peaks at a task size of 1024, i.e. one task per thread. For all vectorizations, work-first scheduling performs the worst, while DBF are slightly faster than the default scheduler on large task sizes (256+).



(a) Performance



(b) Standard deviation

Figure 6.7: Performance of small problem, work-first scheduling, no spinning

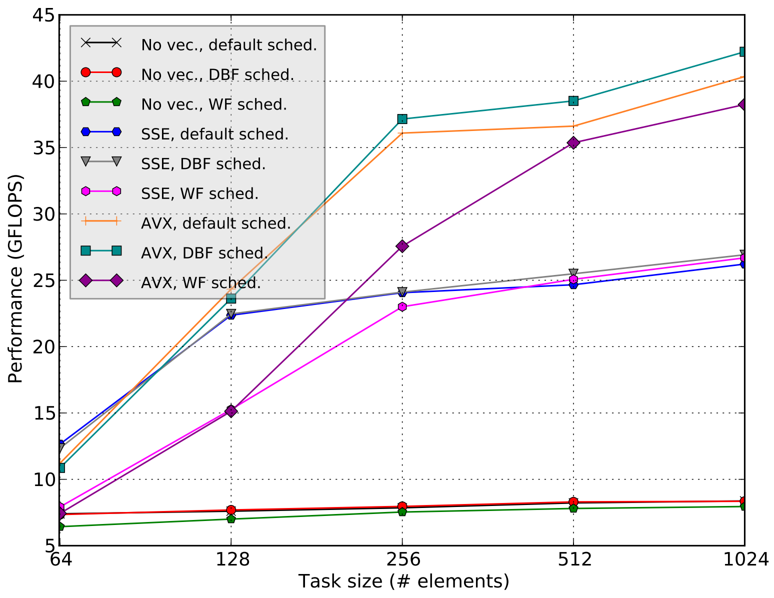


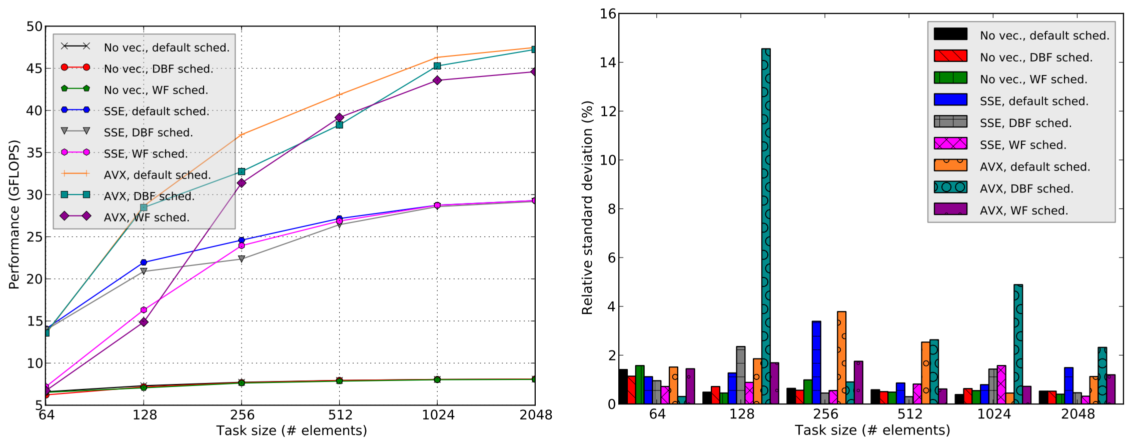
Figure 6.8: Performance with different task sizes, no spinning

6.2 Black-Scholes Results

In this section, performance, energy and energy efficiency results for Black-Scholes are presented. First, performance is tweaked, and results are presented where different task sizes are tried out for different scheduling algorithms on the maximum number of threads. Then, results are presented where the number of threads are varied from 1 through 8, and where the problem size is varied between 2 and 2^{25}. After the performance results, energy and energy efficiency measurements are presented.

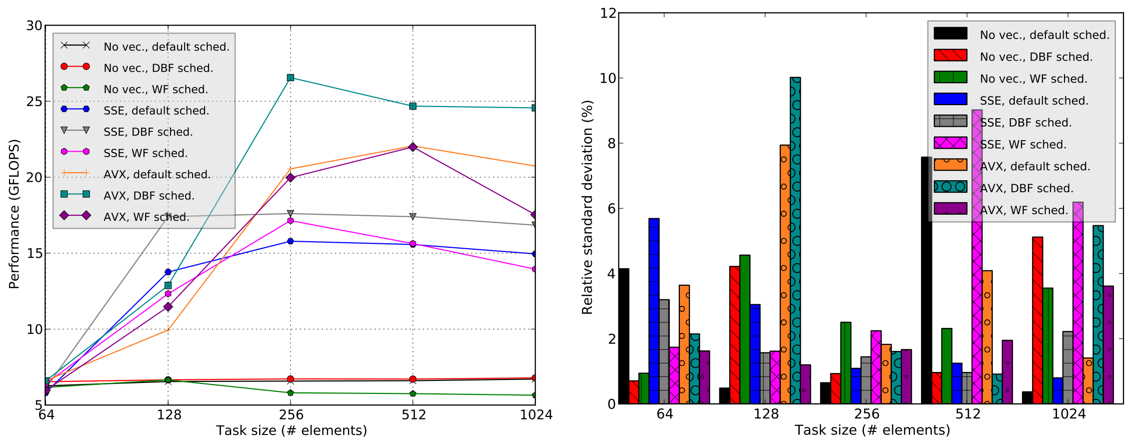
6.2.1 Performance

The results from varying task sizes and scheduling algorithms as described in section 5.2.2 are shown in figures 6.9 and 6.10, and is also tabulated in table B.1.



(a) Performance (b) Standard deviation

Figure 6.9: Performance of large problem with different task sizes, 8 threads



(a) Performance (b) Standard deviation

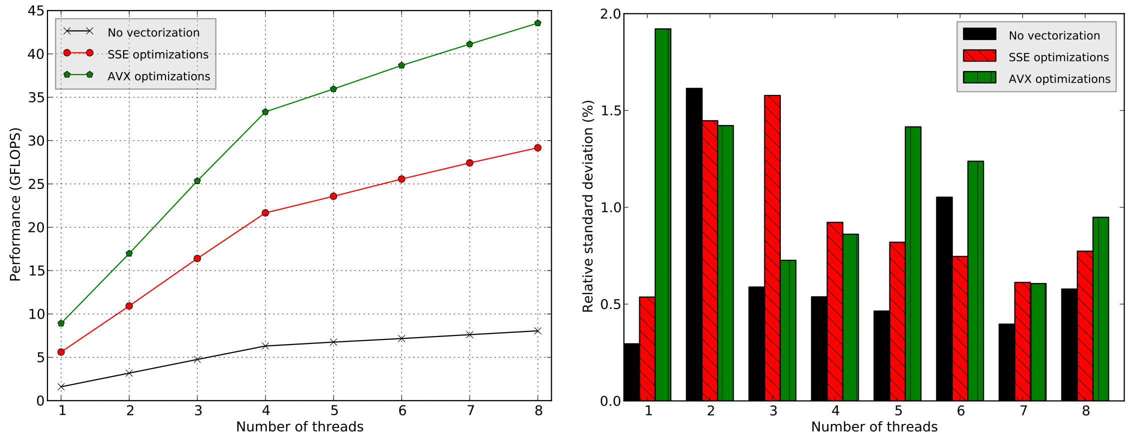
Figure 6.10: Performance of small problem with different task sizes, 8 threads

These results show that having large task sizes gives significantly better performance as long as there are enough tasks. The median computational rate goes from about 14 to 47-48 GFLOPS for AVX, and is doubled for SSE from about 14 to about 28-29 GFLOPS when going from 64 to 2048 on the large problem. On the small problem size, the task size seems to be optimal at 256 for the default scheduler. As a contrast, section 6.1 shows an optimal task size of 1024 for the old Nanos++ version.

For the large problem where there is an abundance of tasks available, it is apparent that as task size increases, the scheduling algorithm has less impact. In fact, for SSE and non-vectorized codes, all three scheduling algorithms converge at task size 2048.

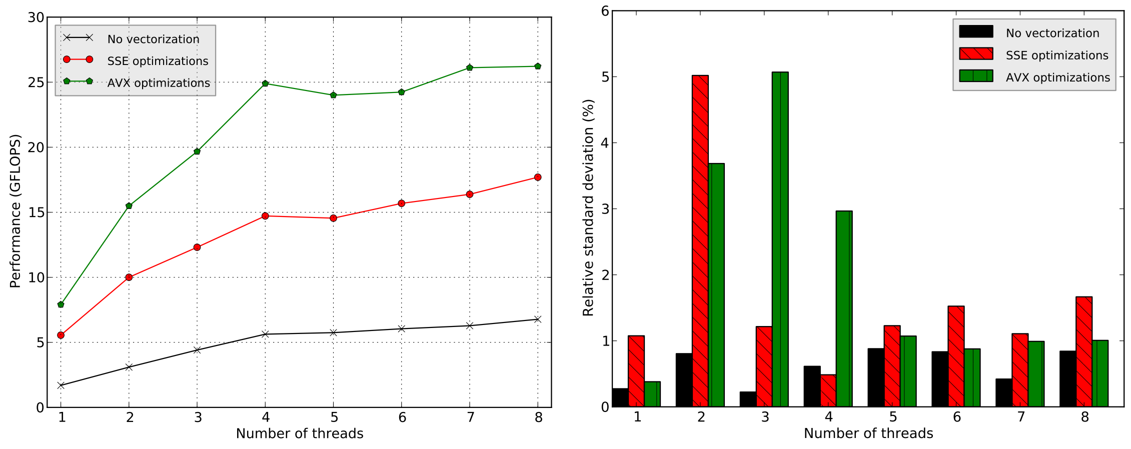
The default scheduler appears to give the best performance for most or all other task sizes, both for SSE and AVX. For the small problem, however, DBF scheduling performs better as long as the task size is 256 elements or more. Work-first shows the lowest performance of the three scheduling algorithms.

The performance tests for one through eight threads were run with the larger task sizes that were found to give the best performance for eight threads using DBF scheduling, as this gave clearly the best performance on the small problem, and very close to the best performance on the large problem. For the large problem, the task size 2048 was chosen. For the small problem, 256 was chosen. The results are shown in figure 6.11 and 6.12, and tabulated in tables B.2 and B.3.



(a) Performance (b) Standard deviation

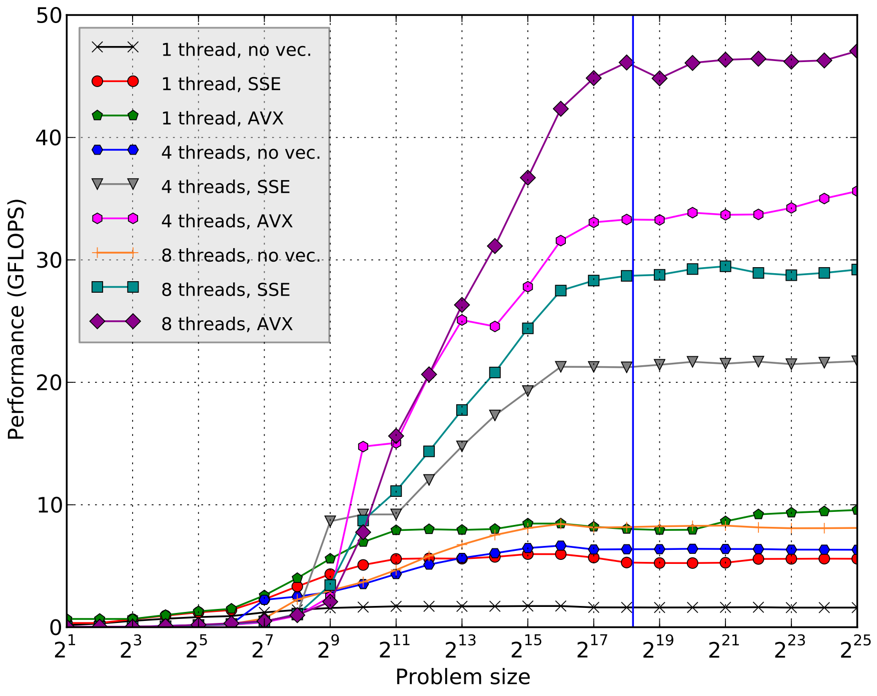
Figure 6.11: Performance of large problem with task size 2048, DBF scheduling



(a) Performance (b) Standard deviation

Figure 6.12: Performance of small problem with task size 256, DBF scheduling

We see a near perfect speedup up to four threads for the large problem. After that, we start assigning more than one thread per core, so the speedup becomes sub-linear. The small problem size also shows close to linear speedup up to four threads, reaching 25 GFLOPS. After that, when hyper-threading comes into effect, the performance decreases at five and six threads, before increasing again on seven and eight.



(a) Performance

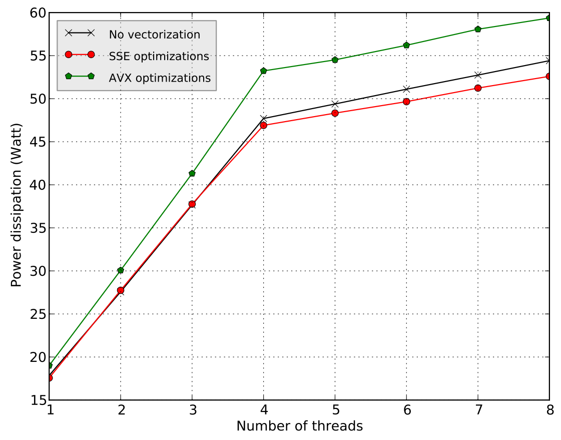
Figure 6.13: Performance of Black-Scholes at different problem sizes

Figure 6.13 shows the performance of Black-Scholes with regards to the problem size. The tabulated data can be found in tables B.4, B.5 and B.6. Again, distributed breadth-first scheduling is used, and task sizes are set so that for $2 \leq N \leq 32$, task size is 8; for $64 \leq N \leq 2048$, task size is $N/8$; for $4096 \leq N \leq 8192$ the task size is 256; for $N = 16384$, task size is 512; for $N \geq 32768$ the task size is 2048. The blue line marks the point where the memory footprint reaches 8MB, i.e. the size of the last-level cache. We here see a slight performance peak at $N = 2^{18}$, where the problem is just small enough to fit in the L3 cache. We also see that hyper-threading using eight threads gives a significant performance gain of about 30-35%.

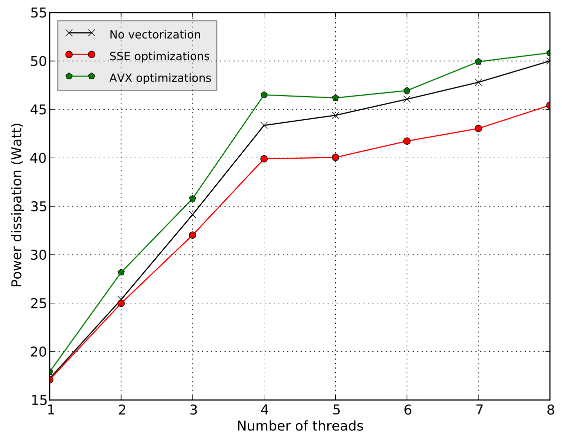
6.2.2 Energy efficiency

In this section we present energy and energy efficiency measurements for Black-Scholes. Three metrics are considered: Power, to see what makes the processor consume more energy; GFLOPS/W, to see the sustained energy efficiency in terms of work done per unit of energy spent, and the energy-delay product, which takes the running time as well as the total energy usage into account.

Figure 6.14 shows the average power dissipation in watts (J/s) of the large and small fixed-size problem for Black-Scholes. The power usage grows more or less linearly up to four cores, which is as expected as each core is identical to the others. After hyper-threading comes into effect at five or more cores, power dissipation continues to steadily



(a) N=2^{25}



(b) N=2^{13}

Figure 6.14: Power dissipation with varying number of threads, Black-Scholes

increase. This shows that hyper-threading is not free energy-wise. One question that remains is if the increased performance from these extra threads is high enough to make hyper-threading more energy efficient than no hyper-threading.

Interestingly, the power dissipation for the SSE implementation is lower than that of the scalar one, possibly because the higher computation rate allows the CPU to sleep more while waiting for memory. The AVX implementation consumes energy at a slightly higher rate. Power dissipation follows the performance curves in this application if we ignore the differences between different vectorizations.

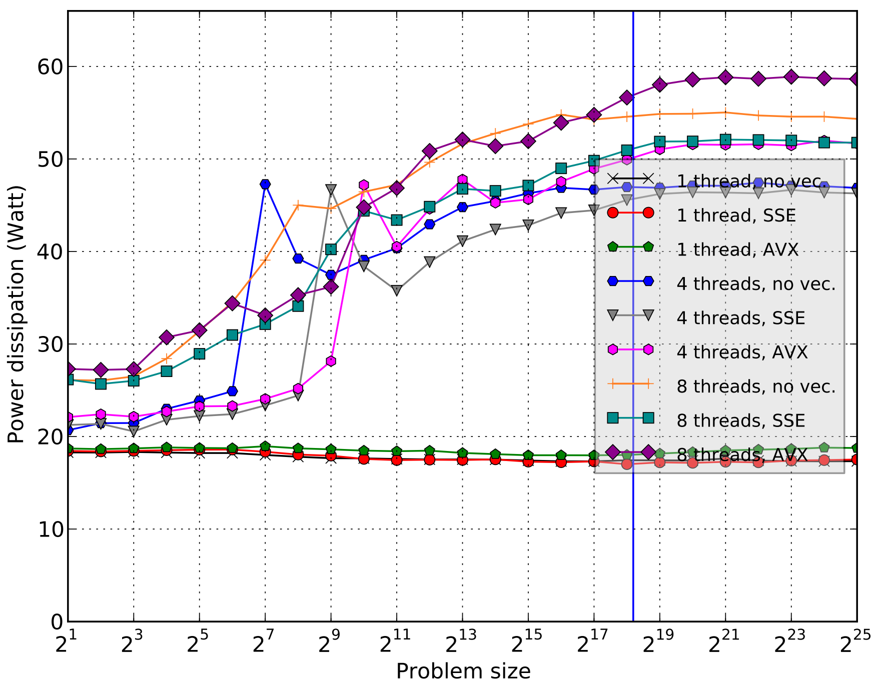


Figure 6.15: Power dissipation, Black-Scholes

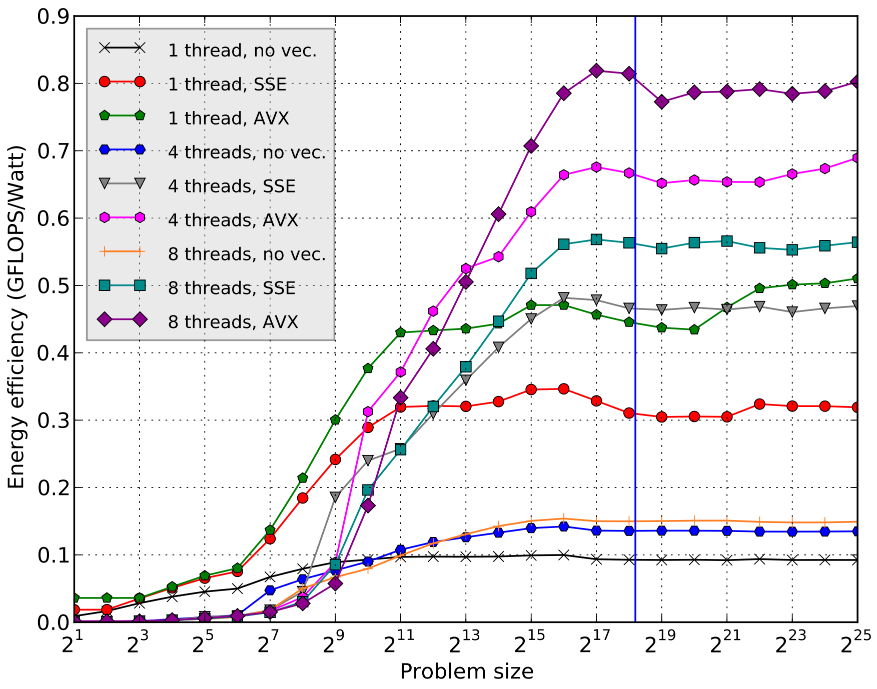
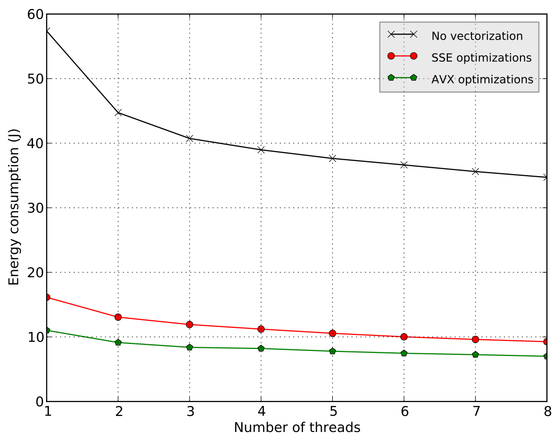


Figure 6.16: GFLOPS/W, Black-Scholes

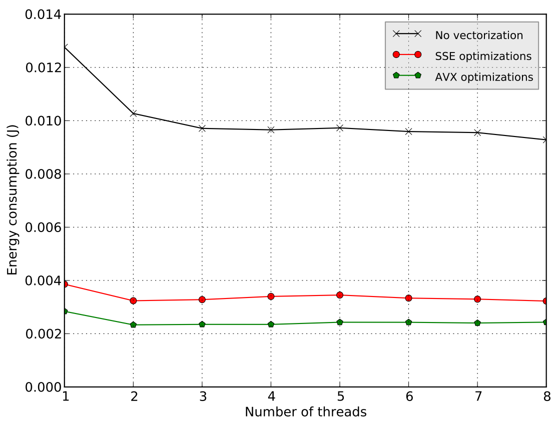
In figure 6.15 the power dissipation of Black-Scholes is shown over a range of problem sizes. Over the whole range, the SSE code has about the same, or sometimes even smaller, power dissipation compared to the non-vectorized codes. The AVX code however has a higher power dissipation, up to about 10% more than the non-vectorized and SSE codes in some places, in particular in the larger problem sizes. It is here that AVX also shows the greatest speedup, however, so it may still be energy efficient to use AVX. In the energy efficiency plot in figure 6.16, it is clear that eight threads with AVX is most energy efficient using this metric. In fact, even a single thread with AVX is approximately as energy efficient as four threads with SSE, and significantly better than a single thread with SSE.

Figure 6.17 shows the total energy spent in joules for Black-Scholes. These plots show that if only looking at energy consumed, running on all cores with hyper-threading is optimal. Vectorization is also beneficial, especially going from no vectorization to SSE, which reduces the energy consumption to about 1/3. Although AVX consumes the least energy, the difference is smaller than figure 6.16 indicates giving only a 25% decrease in energy consumption at eight threads even though performance is significantly better.

Figure 6.18 shows the normalized energy-delay products (EDP) and energy-delay squared products (EDD) for Black-Scholes. Note that the plots have a logarithmic scale. For compactness, the plots for the large and small problem sizes are presented in the same plot. The EDP and EDD are normalized to the EDP and EDD of the single-threaded non-vectorized version for the two problem sizes. Here, the benefit of AVX is more clear compared to figure 6.17, since the time it takes to complete the calculation (the delay) of the output array is taken into account in addition to the energy usage.

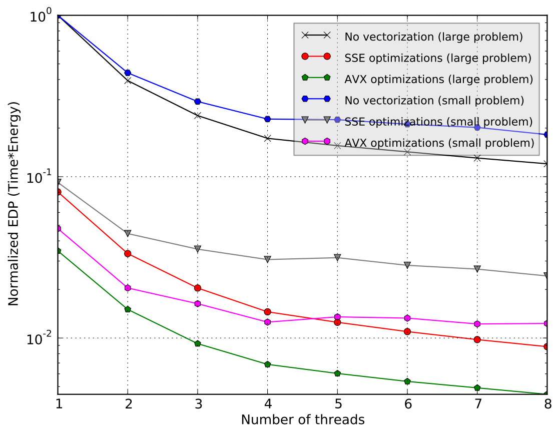


(a) Energy consumed, N=2^{25}

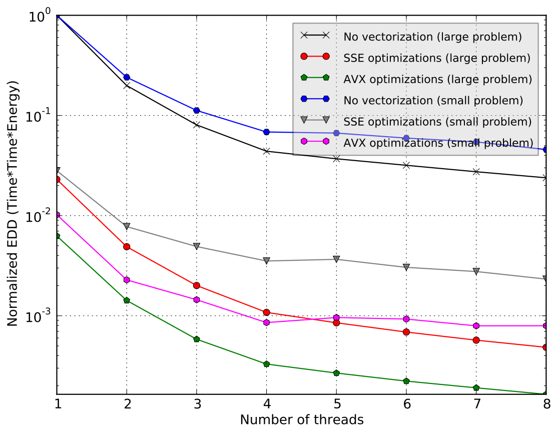


(b) Energy consumed, N=2^{13}

Figure 6.17: Energy consumed for the whole problem, Black-Scholes



(a) Energy-delay products



(b) Energy-delay squared products

Figure 6.18: Normalized energy-delay products, Black-Scholes

With respect to the EDP and EDD metric, it is clear that for the large problem, multi-threading is energy efficient also when including hyper-threads, reducing the EDP by about 70% for the vectorized codes going from one to eight threads. For the small problem, there appears to be little to no benefit from hyper-threading for the AVX implementation, and only a slight improvement in the EDP for SSE. However, up to four threads, the EDP is reduced by about 74%. The EDD product does not appear to reveal any more information compared to what the EDP did for this application.

For the large problem, the EDP is reduced by 99.55% going from non-vectorized single-threaded code to AVX with eight threads. For the small problem, the EDP is reduced by 98.77%.

6.2.3 Discussion

The small decrease in performance in figure 6.13 after the 8MB memory footprint point reveals that Black-Scholes is slightly memory bandwidth sensitive; this is not surprising as the kernel has a constant amount of data reuse per data element, independent of the problem size. In fact, the ratio of flops to memory reads+writes is a constant $153/7 \approx 22$ with the assumptions in table 5.10 in section 5.3 because the algorithm accesses seven arrays per iteration. Comparing this to e.g. a level 1 BLAS-operation like SAXPY, $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$ which has a ratio of $2/3$ (assuming that the scalar α is kept in a register), this number is fairly high, but does not come close to e.g. matrix multiplication, which can easily apply blocking to decrease the number of slow memory accesses as described in chapter 3.

Comparing the figures 6.8 and 6.10, we see a significant degradation in performance when using Nanos++ version 0.7a instead of the older 0.6a. Additionally, we see that the preferred task size for 0.7a (256) is smaller than the one for 0.6a (1024). The speedup from using 0.6a instead of 0.7a is about 1.6x. The reason for the degraded performance with this version of Nanos++ is unknown. On larger problem sizes, no performance degradation was observed for Nanos++ 0.7a, likely because it is less sensitive to scheduling due to the abundance of tasks.

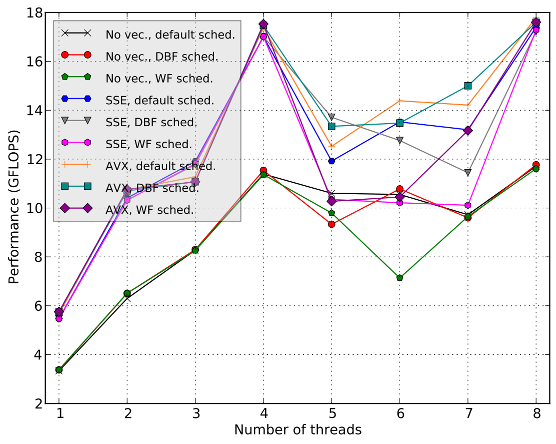
6.3 FFTW

In this section, results for performance, energy and energy efficiency are presented for FFTW using the benchmark described in chapter 4. First, different scheduling algorithms are explored in order to find one that gives good performance. Then performance results using this scheduling algorithm is presented, before presenting the energy measurements through the metrics energy, power, EDP, EDD and GFLOPS/W.

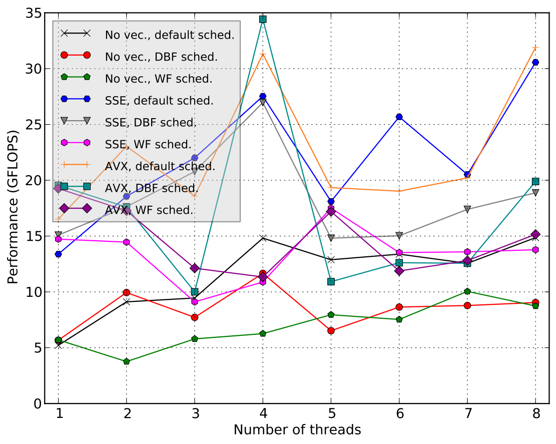
6.3.1 Performance

Figure 6.19 shows the performance of different scheduling algorithms for two fixed problem sizes for 1 through 8 threads. For the large problem size, the difference is relatively minor between different scheduling algorithms, although work-first performs worse than the other two on five and six threads. For the smaller problem size, the default depth-first scheduling algorithm has the highest performance for 2 through 8 threads, with the exception of two threads without vectorization, and four threads with AVX. Based on these results, the default scheduling algorithm is used in subsequent experiments.

In figure 6.20 we see the performance results from FFTW for multi-threading with a large problem size. We see that the benefit of doing more computations per cycle by using vector extensions are not as significant for FFTW as it was on Black-Scholes on this problem size. Only a slightly improved performance is apparent on this problem size when using AVX instead of SSE. This shows that the performance of FFTW is bound by the memory bandwidth when the data does not fit inside the caches. Performance do

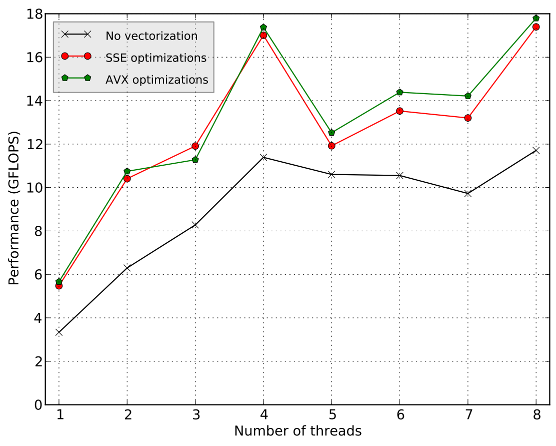


(a) N=2^{25}

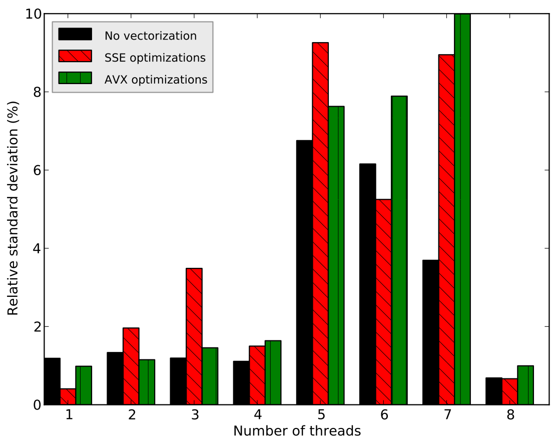


(b) N=2^{14}

Figure 6.19: Performance with different scheduling algorithms, FFTW



(a) Performance



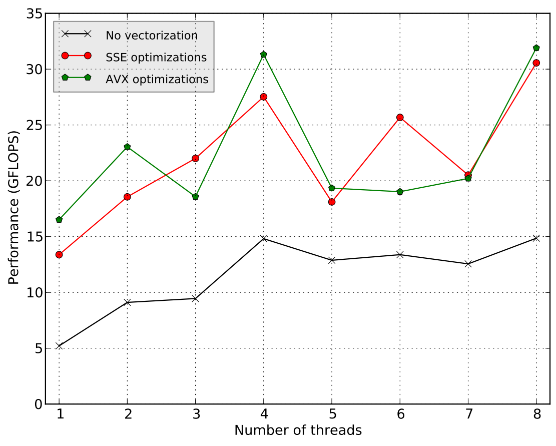
(b) Standard deviation

Figure 6.20: Performance with regards to number of threads, FFTW, N=2^{25}

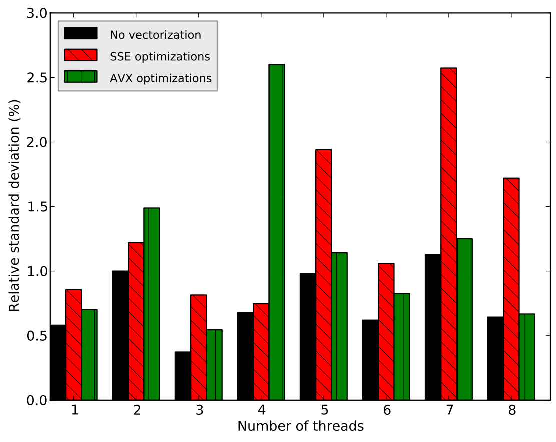
increase significantly when increasing the number of threads up to four threads however, which may be due to better utilization of memory bandwidth as more memory requests can be done in parallel.

Figure 6.21 shows the performance of FFTW for a smaller problem size which fits in the L2 cache. The performance in terms of GFLOPS is significantly higher compared to the large problem size. We see a significant increase in performance from using vectorized codes; in particular, we see a benefit of the larger vectors in AVX when using 1, 2, 4 or 8 threads. For non-2^n number of threads, the performance is degraded; this was also apparent on the large problem size. We attribute this to the nature of FFTW which has the highest performance when N is a power of two [78, sec. 4.3.1 and 4.3.3], and that power-of-two number of threads divides the problem evenly.

Figure 6.22 shows the performance as the problem size increases for one, four and eight threads. The tabulated data can be found in tables B.9, B.10 and B.11. For all

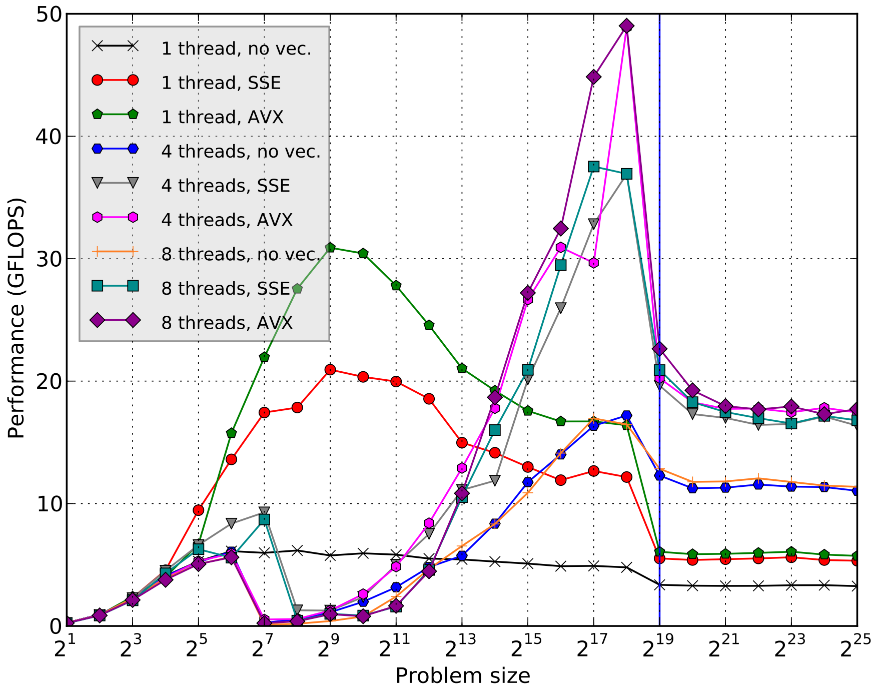


(a) Performance



(b) Standard deviation

Figure 6.21: Performance with regards to number of threads, FFTW, N=2^{14}



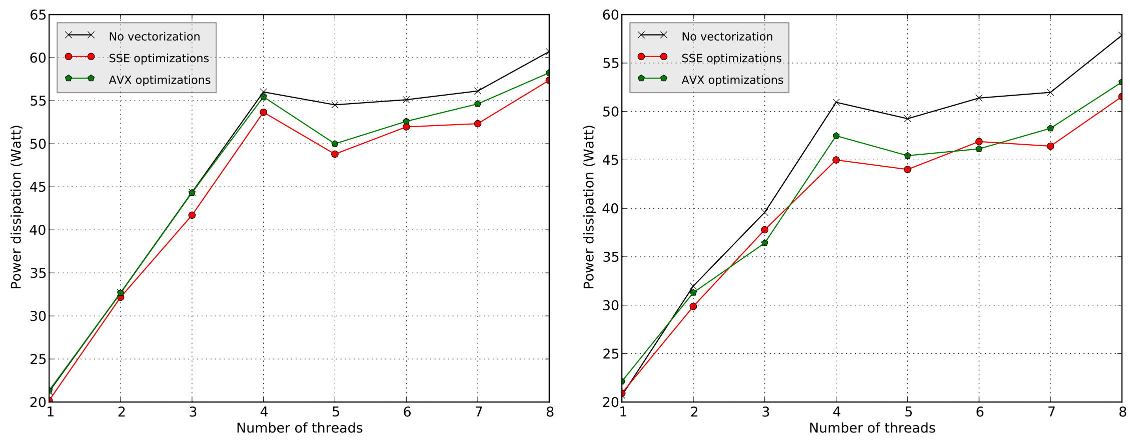
(a) Performance

Figure 6.22: Performance of FFTW at different problem sizes

curves, we see a large dip around $N = 2^{19}$. For two arrays of complex single precision floats, a problem size of $N = 2^{19}$ would consume exactly 8MB of memory, which is exactly the size of the L3 cache of the system the benchmarks are run on. A blue line marks the problem size with a $8MB$ memory footprint.

We also see a dip for the smaller problem sizes for all multi-threaded experiments around $N = 2^6$ and $N = 2^7$. We explain this with that the FFTW planner found it more efficient to run on a single thread up to $N = 2^6$. Once FFTW runs on multiple threads, overheads like task creation and scheduling becomes evident.

6.3.2 Energy efficiency



(a) N=2^{25} (b) N=2^{14}

Figure 6.23: Power dissipation with varying number of threads, FFTW

Figure 6.23 shows the average power usage in watts (J/s) of the fixed-size problems for FFTW. As with Black-Scholes, the power usage grows linearly up to four threads, as expected, since each new thread activates another core. After hyper-threading comes into effect at five or more cores, no definite increase in power dissipation is seen before eight threads. We attribute this to the degraded performance for five, six and seven threads seen in figures 6.20 and 6.21.

As a contrast to what was seen in figure 6.14 where the power dissipation curves follows the performance curves to a large degree, the fluctuations in performance when going from four to five threads, or three to four, is not as pronounced for FFTW. We explain this with that non-power-of-two number of threads give a less optimal subdivision of the problem, so that more work is done in total to compute the FFT. This causes each core to do as much work per second as before, but also do more work in total.

An interesting result is that the non-vectorized code uses more energy than the vectorized one for the large problem in FFTW. One explanation for this is that the activity in the cores are less for the vectorized codes. FFTW is highly dependent on memory bandwidth as we see in figure 6.22, and vectorization may simply cause the processor to be less active as it can do all the work it needs to do in less time, and then go back to waiting for memory.

In figure 6.25 the power dissipation of FFTW is shown over a range of problem sizes. As in figure 6.23, the non-vectorized code has the same or higher power dissipation as SSE, especially on four and eight threads. In fact, on many problem sizes, the AVX code has a lower power dissipation. Looking at the energy efficiency in figure 6.25, the single-threaded results show a significant peak at $N = 2^9$. At that problem size, the whole input and output arrays fit in the L1 cache, which may explain this peak. With AVX, this gives a high GFLOPS rate as the memory bandwidth is very high, while the power dissipation is fairly low, at little more than 20W.

Looking at the energy consumption in figure 6.26, we see little to no benefit from

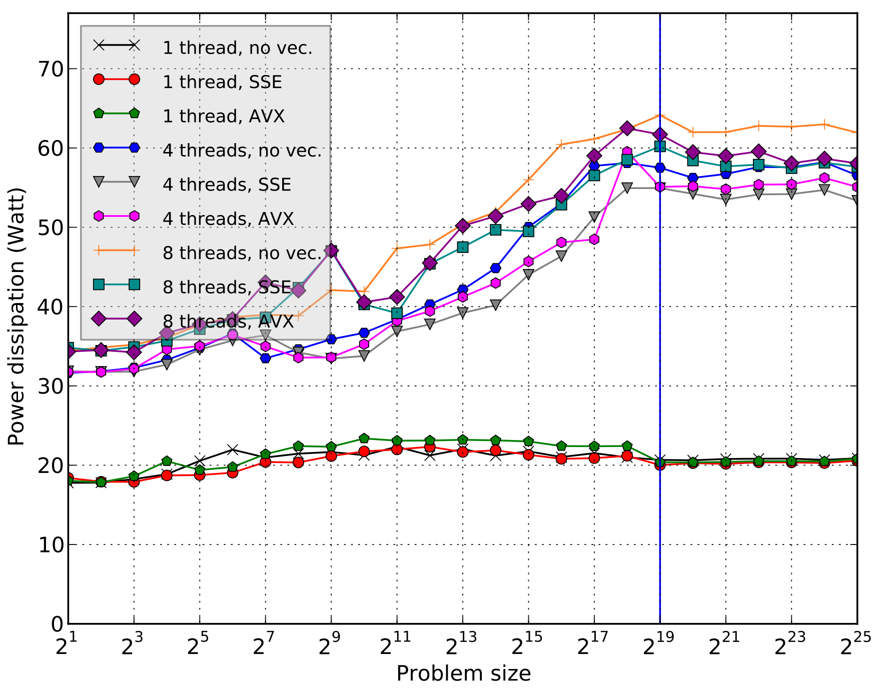


Figure 6.24: Power dissipation, FFTW

multi-threading beyond two threads when it comes to total energy spent to compute the FFT. Additionally, the AVX implementation appears to use about the same amount of energy as the SSE one, with the exception of three threads where SSE uses less energy. For the small problem the single-threaded version spends the least energy; this is backed up by figure 6.25. Here we also see that for single-threaded execution, AVX is the most energy efficient choice if we consider the pure energy metric.

Figure 6.27 shows the normalized energy-delay product (EDP) and energy-delay squared product (EDD). Here we see that for the large problem, the EDP is reduced by about 75% with four threads compared to one thread without vectorization, which shows that multi-threading gives a definite improvement in energy efficiency with regard to the EDP. However, no added benefit is gained from hyper-threading. There is definite improvement in energy efficiency also for the vectorized codes, with about 70% for four cores versus single-threaded.

The total energy consumption for more than one thread for the small problem with SSE or AVX is higher than for a single thread. However, the EDP shows a definite improvement of about 40-50% when going from one thread to four. Going from single-threaded non-vectorized code to four threads vectorized code gives a 90.36% reduction for the large problem, and 93.65% reduction for the small problem.

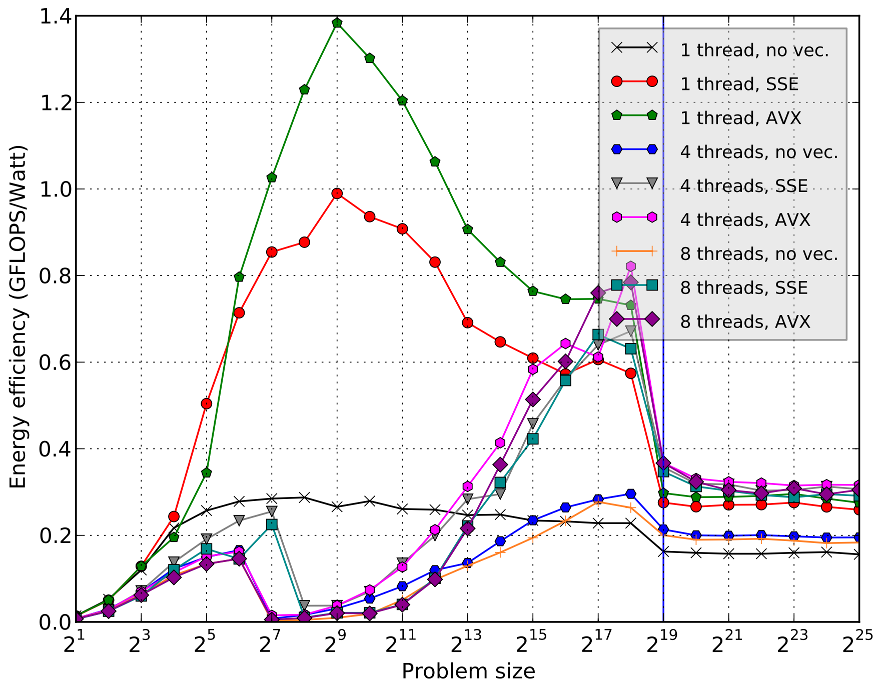
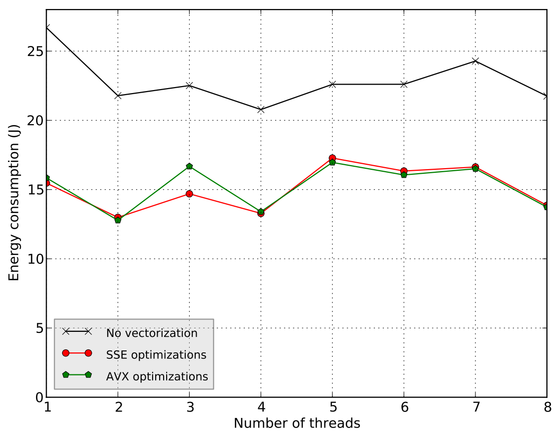
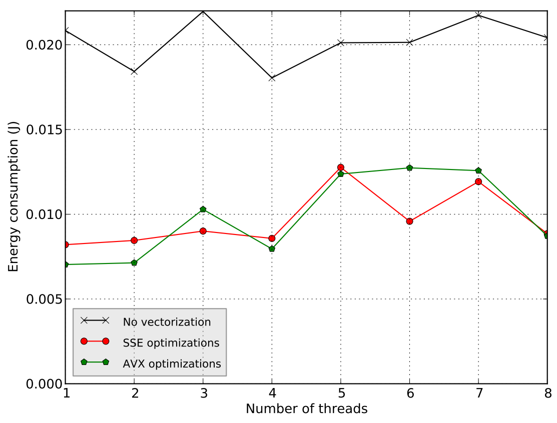


Figure 6.25: GFLOPS/W, FFTW

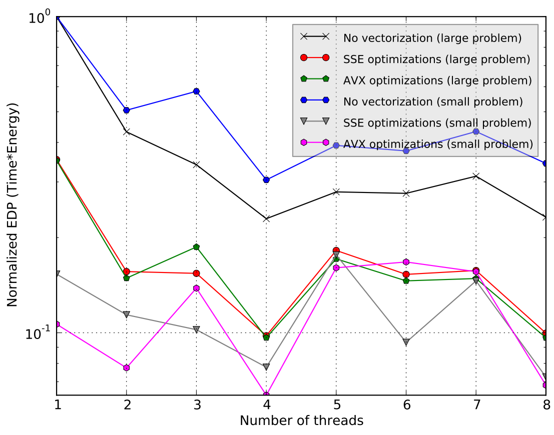


(a) Energy consumed, N=2^{25}

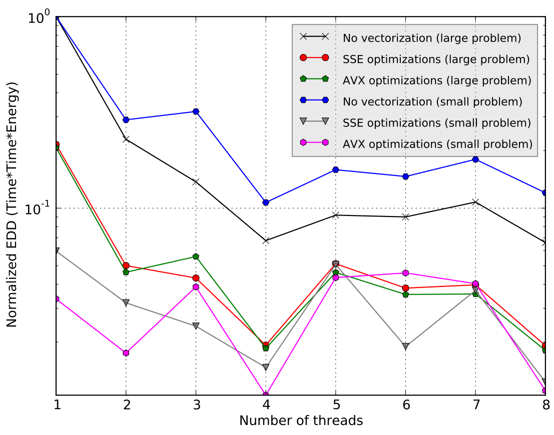


(b) Energy consumed, N=2^{14}

Figure 6.26: Energy consumed for the whole problem, FFTW



(a) Energy-delay products



(b) Energy-delay squared products

Figure 6.27: Normalized energy-delay products, FFTW

6.3.3 Discussion

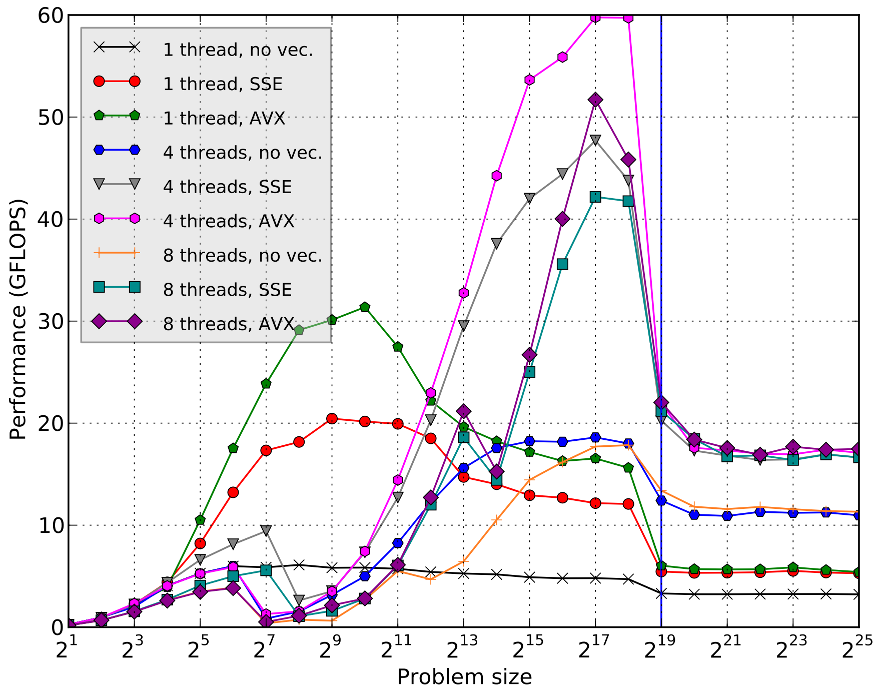


Figure 6.28: Performance for various problem sizes, FFTW, Nanos++ 0.6a

As with Black-Scholes, running FFTW on version 0.6a of Nanos++ gives better performance, particularly on the smaller problem sizes. The performance over different problem sizes is shown in figure 6.28. We see a peak at about 60 GFLOPS, which is about 10 GFLOPS higher than with 0.7a.

6.4 Matrix multiplication

In this section, results for performance, energy and energy efficiency are presented for matrix multiplication. Due to the large task sizes, scheduling was not found to have any significant impact; therefore, the default scheduling algorithm is used for this application.

6.4.1 Performance

The performance for matrix multiplication was measured, and the results are presented in figure 6.29 and tabulated in table B.12. The block size for each task was chosen to be greatest value of b so that $32 \leq b \leq 1024$, and $(N/b)^2 \geq$ # threads. This ensures that there are enough tasks available for all threads, as there may at any point exist up to $(N/b)^2$ tasks that can be performed independently, as there are only $(N/b)^2$ blocks in C which is shared.

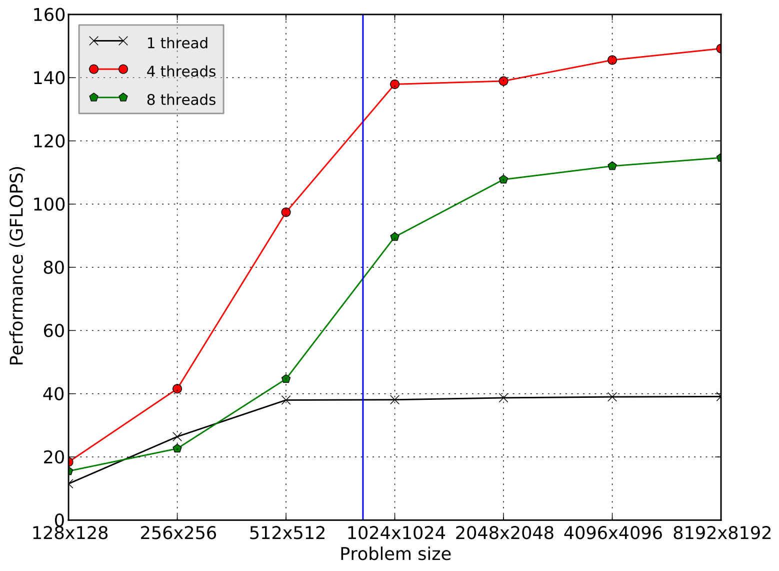


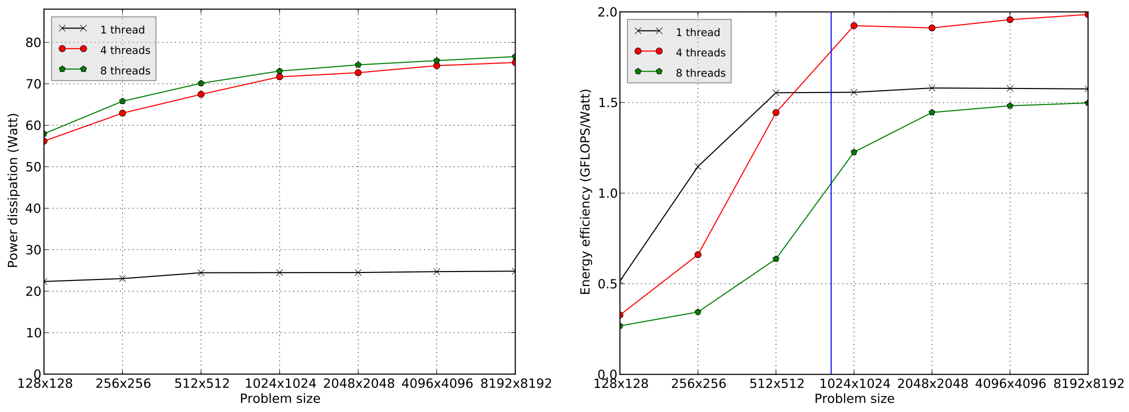
Figure 6.29: Performance and standard deviation for matrix multiplication

The peak performance is significantly higher on matrix-matrix multiply than both Black-Scholes and FFTW. This is expected because of the high reuse of the data in cache in the blocked matrix-matrix multiply algorithm, in addition to ATLAS' autotuning which chooses the optimal block sizes and other parameters based on benchmarks of the system during installation. Matrix-matrix multiply can get close to the peak performance of the CPU if an efficient algorithm is used because of this high reuse. Running with more than four threads, i.e. hyper-threading, shows to have a negative impact on performance.

Multi-threading with four threads always gives a speedup over one core, which is expected. The speedup becomes higher the larger the matrices become. Why hyper-threading has a negative effect is likely due to two threads sharing a cache. This results in a lower reuse because smaller blocks must be used in blocked matrix-matrix multiply. Additionally, because parallelization is introduced outside of the ATLAS library, cache interference will occur because ATLAS will assume that it has 32kB of L1 cache available, while in reality this cache is shared. Components like the ALU is also shared between hyper-threads, and since matrix multiplication is mostly bound by the computational speed of the CPU and not memory bandwidth, hyper-threading would likely not benefit this application much even if each thread had their own cache.

6.4.2 Energy efficiency

In this section, energy and power measurements and energy efficiency measurements are presented for matrix multiplication using the four metrics power, energy, GFLOPS/W, energy-delay product and energy-delay squared product.



(a) Power dissipation (b) Energy efficiency

Figure 6.30: Power dissipation and energy efficiency, matrix multiplication

In figure 6.30 the power dissipation of matrix multiplication is shown over a range of problem sizes. The results show that hyper-threading does not use a significant amount of extra energy. This is can be compared with the performance results, where running on eight threads gave lower performance. With hyper-threading, the threads on one core share the ALU, so the power dissipation does not increase since the application is compute bound.

With regards to energy efficiency in GFLOPS/W, figure 6.30 shows that for matrix multiplication, hyper-threading with eight threads performs worst if comparing with one and four threads. Interestingly, four threads perform better than a single thread for all problem sizes, while single-threaded execution are optimal with regard to this metric up to matrices of size 512×512. After this, four threads is shown to perform significantly better.

Figure 6.31 shows the normalized total energy consumption for matrix multiplication. The numbers were normalized to the single-threaded baseline in order to present all the numbers in the same graph. Here we see that with eight threads, more energy is consumed in total for all problem sizes than both four threads, and one thread. As with the plot in figure 6.30, using a single thread seems to be the most efficient energy-wise up to and including the problem size 512×512.

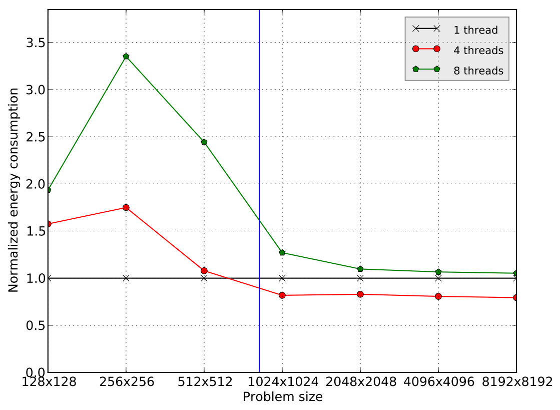
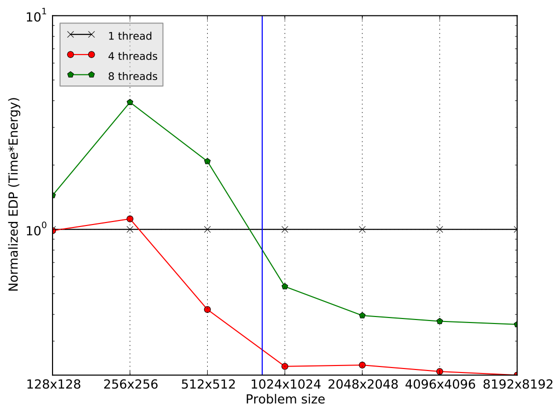
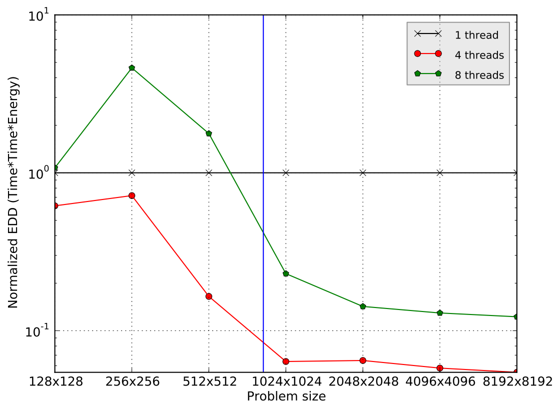


Figure 6.31: Normalized energy consumption, matrix multiplication



(a) Energy-delay product (EDP)



(b) Energy-delay squared (EDD)

Figure 6.32: Normalized energy-delay products, matrix multiplication

Figure 6.32 shows the normalized energy efficiency as given by the energy-delay (EDP) product and energy-delay squared (EDD) products. Note that the y-axes are logarithmic. Using the EDD metric, four threads is shown to be the most energy efficient for all problem sizes, while eight threads is more energy efficient than a single thread for large problems. For the largest problem size, the EDP is reduced by about 80% when going from one to four threads. The EDP and EDD gives different results at the two smallest problem sizes, where EDD, which favors performance, gives a lower value for four than one thread, while the results are opposite for the EDP.

Chapter 7

Performance Modelling and Discussion

In this chapter, we first present two models for the performance and overhead of the benchmarked applications. Then, we present a discussion on the impact of vectorization, hyper-threading and cache misses in the context of energy efficiency.

7.1 Discussion and Analysis of Performance

In this section, we derive two models for performance and overhead. The first model describes the task creation and scheduling overhead in Black-Scholes, and uses performance numbers from the experiments to test the accuracy of the model. The second model provides an estimate of the impact of cache misses in all three applications.

7.1.1 Task creation and scheduling

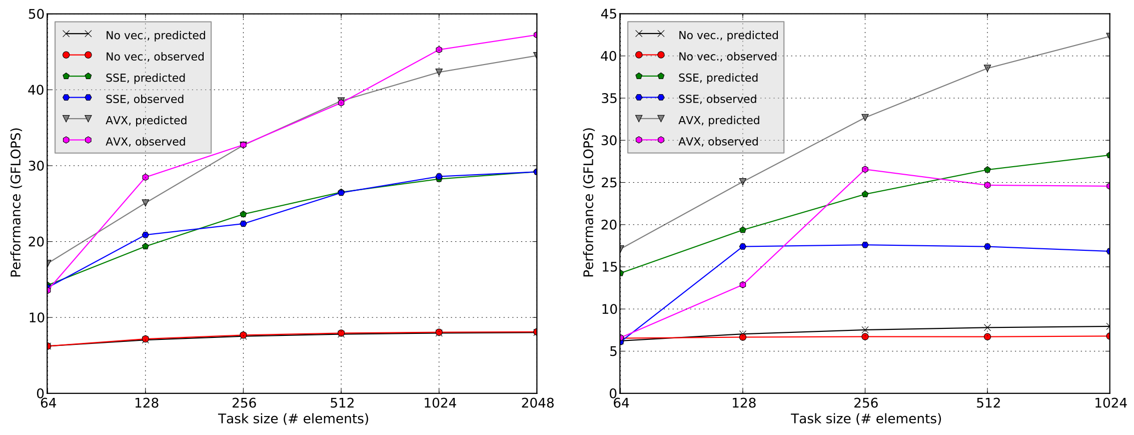
Task scheduling and creation generates extra overhead in the application. The more tasks there is, the more overhead there are, which is visible in the figures 6.9 and 6.10. In this section, an estimate for the task scheduling overhead is estimated by creating a simplified model for the performance for Black-Scholes. Additionally, the impact of the `NX_SPINS` variable in the older Nanos++ version is discussed.

Task overhead estimation

Figures 6.9 and 6.10 gives an indication on the overhead of scheduling tasks. Considering only the overhead from generating and scheduling tasks, and the computation time, a model can be derived where task scheduling and generation cost is quantified. Let T be the total running time, T_c be the computation time, T_s be the scheduling overhead, t be the time required to create and schedule one single task, N be the problem size, and n_t be the task size. Additionally, let P_{max} be the maximum performance that can be achieved given no overhead, i.e. $P_{max} = 153N/T_c$, where 153 is the number of flops per computed element. Then, the following equations hold:

$$T_c = \frac{153N}{P_{max}}$$
$$T_s = T - T_c = tN/n_t$$
$$t = \frac{T_s}{(N/n_t)}$$

P_{max} can be estimated by using a very large task size, so that the task scheduling overhead becomes insignificant. Then, T_c and T_s can be computed, and t can be estimated. The results with DBF scheduling is seen in figure 7.1. The task size used to estimate P_{max} was 16384 on $N = 2^{25}$. The task scheduling overhead per task, t, was computed using the non-vectorized code, and then applied to both AVX, SSE and non-vectorized results, with the assumption that the task scheduling overhead is independent of the task execution time.

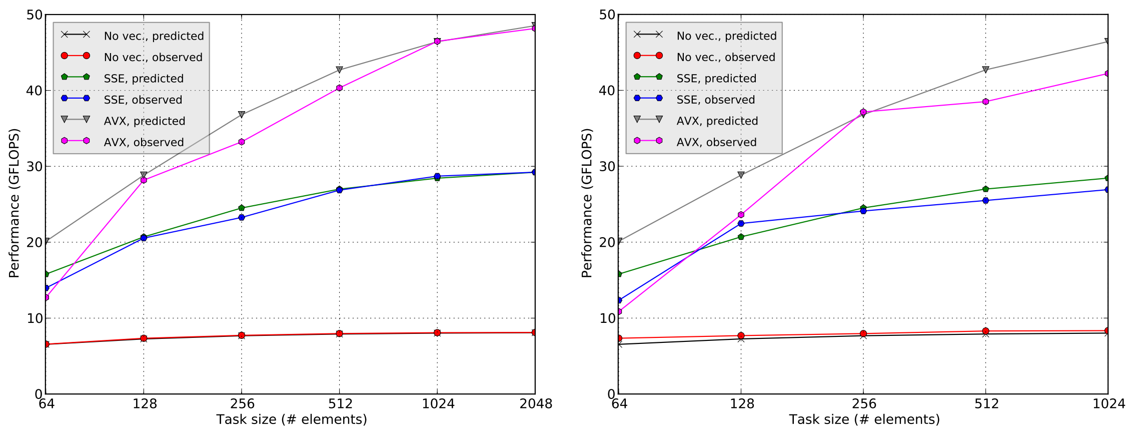


(a) Large problem (b) Small problem

Figure 7.1: Predicted vs. Observed performance in Black-Scholes at different task sizes using Nanos++ v. 0.7a

We see that for the non-vectorized results, the predictions match almost exactly for the large problem, while it slightly over-estimates for the small problem. For the large problem, the predictions match fairly well for the vectorized results although the estimates are slightly lower than the observed performance. For the vectorized results on the small problem size, the predictions are significantly off. We see a predicted performance of about 42.5 GFLOPS for AVX with the largest task size, but only about 25 GFLOPS is observed. However, looking at figure 6.8, it is apparent that using the old version of Nanos++, 0.6a, better performance can be achieved. Using these results using the DBF scheduling algorithm with `NX_SPINS` set to 1, we get the results in figure 7.2.

In figure 7.2, the predictions for task overhead matches more accurately for the small problem. This indicates that the simple performance model described above may be reasonable.



(a) Large problem (b) Small problem

Figure 7.2: Predicted vs. Observed performance in Black-Scholes at different task sizes using Nanos++ v. 0.6a

Impact of NX_SPINS in Nanos++ v. 0.6a

Comparing for instance the figures 6.2 and 6.5, it shows that adjusting the number of times the scheduler spins before having an idle thread yield has a big impact on both performance and variability when going to five or more threads, i.e. having up to two threads per core. Additionally, from figures 6.2, 6.3 and 6.4, it appears that on the default value of `NX_SPINS`, the work-first and distributed breadth-first schedulers have less variability, and perform better than the default scheduler, but on these, setting `NX_-SPINS` to 1 seems to have a less dramatical effect.

The distributed breadth-first and work-first schedulers differ from the default scheduler in that they implement work-stealing; i.e. if a thread does not have any more ready tasks in its local thread pool, it will steal tasks from the other threads. As a consequence, threads will spend less of their time being idle because they will instead simply steal tasks from the other threads, and thus the `NX_SPINS` parameter has less impact. The default scheduling algorithm, however, does not use work stealing, so when a task have completed all its tasks and there are no more tasks in queue the thread becomes idle. The scheduler will spin until more work is found for this thread, or until the Nanos++ scheduler makes it yield.

The reason for spinning having such a large performance impact is still unknown. In Nanos++ 0.7a, setting `NX_SPINS` to 1 has either no effect, or a negative impact on performance.

7.1.2 Cache behavior

The cache miss and hit counts were recorded for the L2 and L3 caches using performance counters as described in section 5.2.3. The number of main memory accesses is assumed to be the same as the number of L3 cache misses, and this is used to estimate overheads due to main memory accesses.

The main memory accesses to flops-ratio is given as

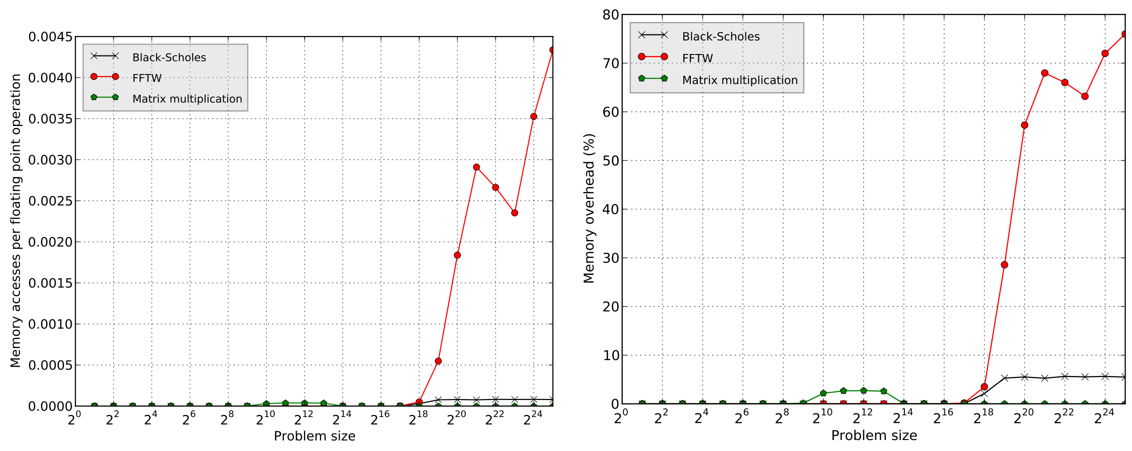
$$R_m = \frac{\text{\# main memory accesses}}{\text{\# floating point operations}}$$

The number of floating point operations is given in table 5.10 for each application. This can be used to find out or confirm why we see such a large drop in performance in some applications like FFTW, but only a slight drop in applications like Black-Scholes, and barely any drop in performance in matrix multiplication.

Given the memory access latency for main memory, we can estimate the time spent waiting using a simple model. Assume that the memory access to flops ratio R_m is given as above, and that the maximum performance for a single core is $P_{1,max}$. Additionally, assume a latency L, in clock cycles, for retrieving a page from main memory, and let F be the frequency on which the CPU operates. Then, the overhead is given as

$$Overhead = \frac{LR_m}{F/P_{1,max} + LR_m}$$

The $F/P_{1,max}$ term gives the number of cycles per flop at peak performance, which potentially can be far less than one due to instruction level parallelism. LR_m is the number of cycles spent waiting for memory per flop. In other words, $F/P_{1,max} + LR_m$ is the number of cycles spent per flop including the computation time and overheads, and $(F/P_{1,max} + LR_m)^{-1}$ is the number of flops per cycle. This model is quite simplified because it ignores latencies from the L1 through L3 caches and other overheads like scheduling overheads.



(a) Main memory accesses per floating point operation (b) Overhead due to main memory accesses

Figure 7.3: Main memory accesses

Figure 7.3 shows the actual main memory accesses per flop, and an estimated overhead per flop using the models above. Memory accesses per flop is also tabulated in table B.13. The number of main memory accesses is the same as the number of L3 cache misses. For the memory access latency, we use an optimistic estimate of two times the L3 latency, i.e. 62 clock cycles. The maximum computation rate for one thread is estimated to 40 GFLOPS based on the results for matrix multiplication. The graphs clearly show that even if the ratio of memory accesses to flops seem modest, for instance 0.0025 for $N = 2^{22}$ for FFTW, the overhead in running time is significant, at about 65%. Black-Scholes shows only a slight overhead of approximately 5% when the problem no longer fits in L3, and matrix multiplication shows an even smaller overhead, of about 2.5%; this is in agreement with the results in chapter 6.

7.2 Energy Efficiency

In this section, the energy efficiency results are discussed. First, the impact of vectorization on energy efficiency is discussed. Then, the results from using multi-threading with or without vectorization, and with and without hyper-threading, is discussed.

7.2.1 Impact of vectorization

The results in section 6.4.2 gives no indication of a higher energy usage when running with SSE as opposed to running the scalar version of the applications. However, the codes vectorized with AVX generally has a 5-10% higher energy consumption than both SSE and non-vectorized codes. Performance-wise, SSE has a potential of up to four times speedup versus non-vectorized codes for single precision, and AVX have a potential of up to eight times. This speedup is often not possible to achieve; neither Black-Scholes or FFTW get speedups of this magnitude, although Black-Scholes is fairly close, with 3.9x speedup for SSE with one thread, and 6.5x speedup with AVX with one thread.

Because of the little extra energy spent, both AVX and SSE is very efficient energy-wise if the application can be efficiently vectorized, like Black-Scholes and FFTW. It is clearly seen in the results for both these applications that both SSE and AVX is far more energy efficient than the scalar codes.

7.2.2 Impact of multiple cores and hyper-threading

Figures 6.14 and 6.23 show a near linear increase in power consumption as the number of threads increase, up to four threads. From four to eight threads, the increase in power consumption is still linear, but with a smaller coefficient. This is reasonable because each core has its own set of registers, cache, instruction fetch/decode units, ALU, and so on, and as such, if there is enough available parallelism, one additional core will become active if the number of threads are increased by one, giving a constant increase in power consumption.

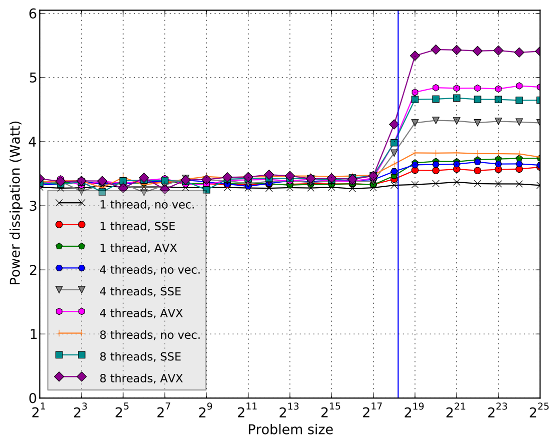
As a contrast to vectorization, which uses only a minimal amount of extra energy, activating another CPU core gives a definite increase in power consumption. In order for this to be energy efficient, the speedup must make up for the added energy costs. There are a few ways this can be true.

- A CPU always uses some energy, even when it is not doing any work. Adding more cores will make this fraction smaller.

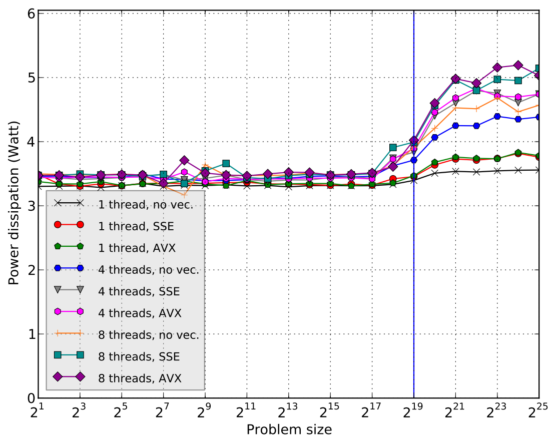
- With more CPU cores, bigger problems can fit in the caches closer to the cores, increasing performance and most likely also energy efficiency.
- The system as a whole, for instance hard drives, graphics cards, various IO-systems, the memory system, etc. uses energy. Again adding more cores will make this fraction smaller. Full system energy measurements are out of scope of this thesis.

7.2.3 Energy usage for large problems

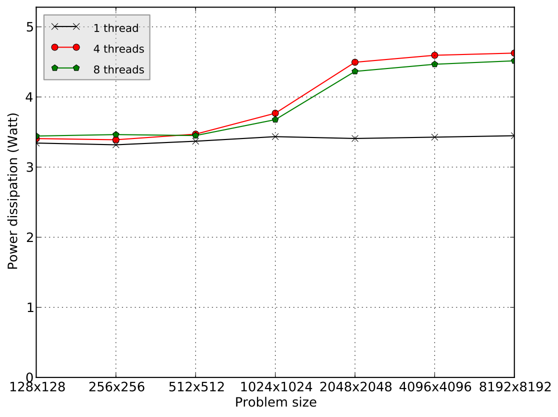
Computing problems that does not fit in the CPU's cache incur additional energy costs because data must be fetched from main memory. Even though the energy of the system as a whole is not measured in this project, the memory controller is located on the processor chip, and thus it is possible to measure some of the extra energy spent due to main memory accesses. Figure 7.4 shows these results. The results are derived by subtracting the power plane 0 (PP0) and 1 (PP1) from the CPU package energy.



(a) Black-Scholes



(b) FFTW



(c) Matrix multiplication

Figure 7.4: Power usage without cores

For all three applications the pattern is similar. Once the problem no longer fits in cache, we see an increase in off-core power dissipation. The high-performance configurations with AVX and four to eight threads increase the most, with up to about 1.6-1.8W for Black-Scholes and FFTW, and about 1.1W for matrix multiplication. This can be explained with that if there are more threads, more memory requests can be done simultaneously. Additionally, the faster kernels are more limited by the memory bandwidth so they will issue more requests in a shorter amount of time. Matrix multiplication is less memory intensive due to cache blocking, which can explain the lower extra power usage for this application for larger problems.

Chapter 8

Conclusion

Energy efficiency and performance results of three task-based programs, Black-Scholes, FFTW and matrix multiplication, were measured and analyzed. Additionally, the energy efficiency of the Intel Sandy Bridge and ARM Cortex-A9 CPU used in the benchmarks were compared; however, these results with conclusion are presented in appendix A. Energy efficiency was presented using the different metrics, like energy, power, GFLOPS/W and process-normalized EDP and EDD. The parameters task size and scheduling policy were optimized for each application.

A comparison of the ARM and Intel CPU was performed, and is presented in appendix A.

8.1 Scheduling

The scheduling algorithm used for task scheduling was found to have a big significance on performance for Black-Scholes in Nanos++ 0.6a, and also slightly for FFTW, but was not found to be very significant for matrix multiplication. The default scheduler was found to give a very high variation in the performance measurements. However, in Nanos++ 0.7a, the differences were smaller and the performance more stable, which was then used for the rest of the experiments.

A series of scheduling algorithms was explored: Depth-first with and without task stealing (default and work-first algorithms, resp.), and breadth-first with work stealing (distributed breadth-first). Work-first was found to perform worse than distributed breadth-first or the default depth-first scheduler for all applications, while the default and DBF scheduler differed only slightly. For FFTW, the default scheduler was found to be slightly superior, while on Black-Scholes, DBF was found to perform better. Due to the large tasks in matrix multiplication, scheduling has little impact on that application.

8.2 Vectorization

It was found that vectorization using SSE and AVX on the Sandy Bridge not only is very efficient performance-wise, but also with regards to energy efficiency. The vector instructions were found to consume little to no extra energy per second, while doing considerably more work per cycle. The performance shows a great improvement in Black-Scholes when using vectorization, and a smaller but still significant improvement in FFTW. The total consumed energy is significantly lower and the GFLOPS/W is much higher using vectorization. The EDP and EDD products show a clear benefit from vectorization as well. The performance of Black-Scholes is significantly higher using AVX instead of SSE, but in FFTW the difference was only minor.

8.3 Multi-threading

FFTW showed the best performance when the number of threads was 1, 2, 4 or 8, with a maximum at four threads. Black-Scholes got a significant increase in performance from hyper-threading, while FFTW performed roughly the same with four and eight threads. Matrix multiplication performed worse with eight threads than with four.

Multi-threading was found to give a smaller energy-delay product for all three applications, even where the GFLOPS/W peaks at single-threaded execution. For Black-Scholes, the EDP and EDD was at a minimum at eight threads with AVX. For FFTW and matrix multiplication, the EDP and EDD is at a minimum at four threads. For FFTW when $N = 2^{14}$, the optimal with regards to GFLOPS/W was to run with one thread with AVX, however both two, four and eight threads showed a better EDP and EDD.

8.4 Derived models and discussions

The performance of Black-Scholes at different task sizes was analyzed in order to derive a model for the performance which includes task creation and scheduling overhead. The model was found to be quite accurate, differing from the actual results with only about 10% at most if using Nanos++ 0.6a. However, with Nanos++ 0.7a, while the results for when $N = 2^{25}$ was accurate, the results was significantly off when a smaller problem, $N = 2^{13}$ was used.

From the results, FFTW seemed to be significantly more memory bandwidth-bound than the other applications. By measuring the L3 cache misses using performance counters, a model for estimating the overhead due to main memory accesses was derived. When the input and output arrays do not fit in cache, an optimistic estimate of the overhead due to main memory accesses, ignoring latencies in the L1, L2 and L3 caches, was close to 70% of the total running time. For Black-Scholes, this overhead was close to 5%, and for matrix multiplication it was about half, around 2.5%. This fits fairly well with the observations, which shows only a slight drop in performance for Black-Scholes after the memory footprint reaches 8MB, a significant drop for FFTW, and no noticeable drop for matrix multiplication.

8.5 Conclusion - ARM

Please see appendix A for the conclusion related to the ARM results.

8.6 Future Work

In this section, possible future work and projects are described.

8.6.1 Full system energy measurement

In this thesis, only the energy spent by the CPU is measured. Although the CPU is one of the major components with regard to energy usage, many other factors also play a role. One example is the memory system including the memory modules, I/O devices and controllers, network controllers, and so on. Future work includes measuring the system as a whole using energy measurement instruments like the Yokogawa WT210. Then additional optimizations can be performed which not only considers the CPU and its caches, but I/O, bus and memory activity as well.

8.6.2 Benchmarks on consumer-grade ARM CPUs and Ivy Bridge

The results in this thesis are based on a development version (test chip) of the ARM Cortex-A9 MPCore. It may be interesting to also look at consumer versions of the Cortex-A9 and its successor Cortex-A15 to see if the differences are significant. Additionally, nearing the end of the thesis work, Intel Ivy Bridge was released which is the successor to Sandy Bridge, which promises significantly lower power dissipation for the same performance as Sandy Bridge.

8.6.3 GPUs and accelerators

Although Intel's Sandy Bridge CPUs with AVX does utilize larger vectors than previous generations of vector instructions like SSE or MMX, much larger vector processors exists in the form of GPUs. An example is NVIDIA's Fermi architecture which uses 1024-bit vectors. Additionally, low-power GPUs exists for mobile devices, for instance the ARM Mali series. The low power embedded GPUs are particularly interesting because of the Mont Blanc project which aims to create an energy efficient supercomputer using embedded technology.

8.6.4 Additional applications

In this thesis, only a select few applications are benchmarked. More applications should be explored in order to build knowledge about for instance when multi-threading pays off with regards to energy efficiency, and when it doesn't.

8.6.5 Energy-efficient algorithms

Traditionally, most emphasis when it comes to optimizing algorithms, is performance. Very often, performance and energy efficiency is closely related, but in some cases the optimizations leading to more energy efficient algorithms could impact the performance only minimally, or even negatively. Say, for instance, that there is an algorithm that is significantly bound by memory bandwidth. It seems reasonable that if portions of the data is prefetched, then do all computations in bulks, we have larger periods where the CPU frequency may be scaled down (during loads), and scaled back up during computations, which could save energy.

8.7 Concluding remarks

The goal of this thesis is to evaluate two current microprocessors, the Intel Sandy Bridge Core i7 2600, and the ARM Cortex-A9 MPCore quad-core CPU both with respect to performance and energy efficiency using multi-threaded execution as well as vectorization. Three applications, Black-Scholes, FFTW and dense matrix-matrix multiplication were chosen, and results are presented for these.

Vectorization gives little to no increase in power dissipation, while giving significantly higher performance. Multi-threading, while giving higher performance, also increases energy consumption linearly with the number of cores. Both vectorization and multi-threading gives a lower EDP, so we conclude that both of these methods of parallelization are energy efficient.

In Black-Scholes, hyper-threading gives increased performance and energy efficiency for all the metrics presented in this thesis. For FFTW, little to no improvement is gained from hyper-threading neither for performance nor energy efficiency for any metric presented. In matrix multiplication, hyper-threading negatively affects both performance and energy efficiency compared to four threads.

Single-threaded execution gives better GFLOPS/W on small problem sizes in FFTW and matrix multiplication. However, for FFTW, the EDP is smaller with multi-threading with four threads even for a small problem size like $N = 2^{14}$ despite higher GFLOPS/W for a single thread.

Appendices

Appendix A

Performance and Energy Efficiency Results - ARM

The results for ARM are restricted from publishing, and is therefore given out separately from the report. Please contact the author regarding this chapter.

Appendix B

Tabulated Data

This chapter contains the complete data from the experiments whose results were presented in chapter 6.

	Large problem size			Small problem size		
Task size	**GFLOPS (median)**	**Standard deviation**	**GFLOPS (max)**	**GFLOPS (median)**	**Standard deviation**	**GFLOPS (max)**
			No vectorization			
64	6.224	0.071	6.386	6.537	0.046	6.610
128	7.171	0.051	7.231	6.662	0.276	6.694
256	7.678	0.043	7.721	6.722	0.063	6.843
512	7.938	0.040	7.947	6.713	0.065	6.802
1024	8.055	0.051	8.076	6.788	0.342	6.832
2048	8.104	0.043	8.117	4.522	0.038	4.547
			SSE optimizations			
64	13.940	0.133	14.038	6.102	0.194	6.324
128	20.882	0.487	21.254	17.403	0.274	17.784
256	22.351	0.100	22.451	17.602	0.255	17.986
512	26.437	0.080	26.532	17.400	0.168	17.741
1024	28.576	0.407	28.675	16.843	0.371	17.129
2048	29.185	0.135	29.311	13.909	0.522	14.032
			AVX optimizations			
64	13.574	0.042	13.651	6.596	0.142	6.833
128	28.473	3.963	29.092	12.874	1.325	16.477
256	32.740	0.295	32.934	26.557	0.425	26.843
512	38.277	1.000	39.062	24.678	0.226	25.171
1024	45.280	2.199	46.699	24.563	1.334	25.906
2048	47.238	1.090	48.470	19.046	0.201	19.273

Table B.1: Black-Scholes, performance with eight threads, DBF scheduling, different task sizes

NT	RT	Energy (Joule)	GFLOPS (median)	GFLOPS (mean)	Standard deviation	GFLOPS (min)	GFLOPS (max)
			No vectorization				
1	3.212	57.369	1.599	1.597	0.005	1.588	1.602
2	1.623	44.730	3.174	3.148	0.051	3.040	3.196
3	1.082	40.734	4.748	4.738	0.028	4.671	4.759
4	0.817	38.972	6.303	6.293	0.034	6.240	6.338
5	0.762	37.637	6.750	6.741	0.031	6.692	6.791
6	0.717	36.632	7.162	7.155	0.075	6.960	7.222
7	0.675	35.608	7.610	7.609	0.030	7.566	7.646
8	0.638	34.713	8.050	8.033	0.046	7.952	8.081
			SSE vectorization				
1	0.919	16.135	5.607	5.592	0.030	5.533	5.625
2	0.471	13.061	10.920	10.872	0.157	10.617	11.037
3	0.316	11.917	16.397	16.241	0.256	15.787	16.503
4	0.239	11.208	21.660	21.590	0.199	21.347	21.861
5	0.218	10.557	23.575	23.516	0.193	23.215	23.867
6	0.202	10.014	25.565	25.495	0.190	25.229	25.728
7	0.188	9.608	27.425	27.352	0.167	27.076	27.575
8	0.176	9.258	29.172	29.144	0.225	28.735	29.397
			AVX vectorization				
1	0.580	11.023	8.913	8.904	0.171	8.618	9.115
2	0.304	9.129	16.969	16.957	0.241	16.554	17.353
3	0.203	8.383	25.345	25.333	0.184	25.009	25.584
4	0.154	8.207	33.314	33.203	0.286	32.797	33.677
5	0.143	7.783	35.955	35.908	0.508	35.127	36.460
6	0.133	7.467	38.669	38.551	0.477	37.726	39.230
7	0.125	7.249	41.130	41.131	0.249	40.605	41.468
8	0.118	7.000	43.552	43.669	0.414	43.143	44.327

Table B.2: Black-Scholes performance, N=2^{25}, DBF scheduling, task size 2048

NT	RT	Energy (Joule)	GFLOPS (median)	GFLOPS (mean)	Standard deviation	GFLOPS (min)	GFLOPS (max)
No vectorization							
1	7e-04	0.013	1.693	1.691	0.005	1.680	1.694
2	4e-04	0.010	3.096	3.110	0.025	3.088	3.148
3	3e-04	0.010	4.410	4.407	0.010	4.383	4.420
4	2e-04	0.010	5.629	5.626	0.035	5.592	5.698
5	2e-04	0.010	5.747	5.732	0.050	5.651	5.795
6	2e-04	0.010	6.044	6.021	0.050	5.927	6.073
7	2e-04	0.010	6.279	6.272	0.026	6.219	6.304
8	2e-04	0.009	6.774	6.775	0.057	6.702	6.868
SSE optimizations							
1	2e-04	0.004	5.554	5.527	0.059	5.393	5.579
2	1e-04	0.003	10.001	9.820	0.493	9.049	10.345
3	1e-04	0.003	12.309	12.267	0.149	11.930	12.517
4	9e-05	0.003	14.717	14.734	0.071	14.635	14.880
5	9e-05	0.003	14.546	14.580	0.179	14.293	14.867
6	8e-05	0.003	15.683	15.734	0.240	15.353	16.170
7	8e-05	0.003	16.380	16.352	0.181	16.042	16.697
8	7e-05	0.003	17.695	17.690	0.295	17.300	18.223
AVX optimizations							
1	2e-04	0.003	7.905	7.915	0.030	7.873	7.955
2	8e-05	0.002	15.488	15.134	0.558	14.027	15.658
3	7e-05	0.002	19.666	19.733	1.000	18.464	21.413
4	5e-05	0.002	24.895	24.882	0.738	23.630	25.767
5	5e-05	0.002	23.999	23.899	0.256	23.450	24.181
6	5e-05	0.002	24.234	24.259	0.213	23.888	24.652
7	5e-05	0.002	26.111	26.081	0.258	25.607	26.416
8	5e-05	0.002	26.219	26.182	0.263	25.556	26.576

Table B.3: Black Scholes performance and energy, N=2^{13}, DBF scheduling, task size 256

No vectorization						
N	**GFLOPS (median)**	**Energy (Joule)**	**GFLOPS (median)**	**Energy (Joule)**	**GFLOPS (median)**	**Energy (Joule)**
	1 thread		**4 threads**		**8 threads**	
2	0.164	3e-05	0.009	7e-04	0.009	9e-04
4	0.302	4e-05	0.018	7e-04	0.017	1e-03
8	0.518	4e-05	0.036	7e-04	0.035	9e-04
16	0.692	6e-05	0.103	6e-04	0.084	8e-04
32	0.826	1e-04	0.183	7e-04	0.154	0.001
64	0.909	2e-04	0.259	1e-03	0.263	0.001
128	1.211	3e-04	2.239	4e-04	0.716	0.001
256	1.414	5e-04	2.503	6e-04	2.260	8e-04
512	1.573	9e-04	2.881	0.001	2.986	0.001
1024	1.641	0.002	3.521	0.002	3.691	0.002
2048	1.705	0.003	4.341	0.003	4.673	0.003
4096	1.708	0.006	5.114	0.005	5.813	0.005
8192	1.703	0.013	5.653	0.010	6.747	0.010
16384	1.708	0.026	6.044	0.019	7.527	0.018
32768	1.728	0.051	6.466	0.036	8.091	0.033
65536	1.729	0.100	6.664	0.071	8.433	0.065
131072	1.619	0.215	6.350	0.147	8.137	0.134
262144	1.617	0.433	6.368	0.296	8.175	0.268
524288	1.610	0.871	6.373	0.590	8.238	0.535
1048576	1.614	1.732	6.404	1.183	8.277	1.064
2097152	1.624	3.485	6.388	2.366	8.301	2.132
4194304	1.631	6.847	6.384	4.772	8.150	4.307
8388608	1.602	13.899	6.336	9.546	8.083	8.679
16777216	1.601	27.844	6.333	19.093	8.088	17.322
33554432	1.602	55.562	6.325	38.050	8.106	34.499

Table B.4: Black Scholes performance and energy, DBF scheduling, no vectorization

	SSE vectorization					
N	**GFLOPS (median)**	**Energy (Joule)**	**GFLOPS (median)**	**Energy (Joule)**	**GFLOPS (median)**	**Energy (Joule)**
	1 thread		**4 threads**		**8 threads**	
2	0.342	3e-05	0.019	7e-04	0.020	8e-04
4	0.343	3e-05	0.019	8e-04	0.019	8e-04
8	0.649	3e-05	0.036	7e-04	0.034	9e-04
16	0.944	5e-05	0.069	8e-04	0.077	9e-04
32	1.215	7e-05	0.179	9e-04	0.153	9e-04
64	1.402	1e-04	0.210	0.001	0.260	0.001
128	2.275	2e-04	0.418	0.001	0.472	0.001
256	3.329	2e-04	1.111	1e-03	1.038	0.001
512	4.333	3e-04	8.650	4e-04	3.441	9e-04
1024	5.084	5e-04	9.208	7e-04	8.717	8e-04
2048	5.577	1e-03	9.212	0.001	11.117	0.001
4096	5.620	0.002	12.027	0.002	14.364	0.002
8192	5.596	0.004	14.769	0.004	17.741	0.003
16384	5.743	0.008	17.295	0.006	20.804	0.006
32768	5.970	0.015	19.286	0.011	24.410	0.010
65536	5.967	0.029	21.268	0.021	27.493	0.018
131072	5.688	0.061	21.260	0.042	28.312	0.035
262144	5.282	0.129	21.227	0.086	28.696	0.071
524288	5.243	0.263	21.433	0.173	28.781	0.145
1048576	5.239	0.525	21.676	0.345	29.252	0.285
2097152	5.266	1.053	21.523	0.691	29.479	0.569
4194304	5.573	2.089	21.691	1.374	28.935	1.157
8388608	5.584	4.044	21.484	2.788	28.750	2.323
16777216	5.598	8.008	21.608	5.524	28.937	4.594
33554432	5.589	16.091	21.724	10.940	29.210	9.120

Table B.5: Black Scholes performance and energy, DBF scheduling, SSE optimized

	AVX vectorization					
N	**GFLOPS (median)**	**Energy (Joule)**	**GFLOPS (median)**	**Energy (Joule)**	**GFLOPS (median)**	**Energy (Joule)**
	1 thread		**4 threads**		**8 threads**	
2	0.674	3e-05	0.034	8e-04	0.037	9e-04
4	0.672	3e-05	0.035	8e-04	0.039	9e-04
8	0.675	3e-05	0.035	8e-04	0.038	9e-04
16	0.991	5e-05	0.067	8e-04	0.095	8e-04
32	1.296	7e-05	0.134	9e-04	0.193	8e-04
64	1.505	1e-04	0.198	0.001	0.321	0.001
128	2.593	1e-04	0.403	0.001	0.500	0.001
256	4.007	2e-04	0.936	0.001	0.987	0.001
512	5.592	3e-04	2.499	9e-04	2.080	0.001
1024	6.962	4e-04	14.752	5e-04	7.757	9e-04
2048	7.918	7e-04	15.054	8e-04	15.617	9e-04
4096	8.001	0.001	20.602	0.001	20.650	0.002
8192	7.945	0.003	25.088	0.002	26.323	0.002
16384	8.020	0.006	24.564	0.005	31.133	0.004
32768	8.464	0.011	27.808	0.008	36.714	0.007
65536	8.466	0.021	31.576	0.015	42.342	0.013
131072	8.206	0.044	33.070	0.030	44.849	0.024
262144	8.025	0.090	33.298	0.060	46.124	0.049
524288	7.943	0.183	33.266	0.123	44.827	0.104
1048576	7.955	0.369	33.858	0.245	46.087	0.204
2097152	8.645	0.687	33.682	0.492	46.348	0.407
4194304	9.212	1.320	33.715	0.982	46.432	0.815
8388608	9.353	2.592	34.245	1.930	46.195	1.642
16777216	9.459	5.108	35.011	3.824	46.290	3.261
33554432	9.574	10.117	35.617	7.620	47.056	6.406

Table B.6: Black Scholes performance and energy, DBF scheduling, AVX optimized

NT	RT	Energy (Joule)	GFLOPS (median)	GFLOPS (mean)	Standard deviation	GFLOPS (min)	GFLOPS (max)
No vectorization							
1	1.258	26.694	3.336	3.329	0.040	3.247	3.400
2	0.666	21.781	6.297	6.287	0.084	6.105	6.383
3	0.507	22.511	8.278	8.261	0.099	8.028	8.379
4	0.371	20.779	11.390	11.367	0.126	11.119	11.511
5	0.414	22.600	10.603	10.142	0.685	8.845	10.735
6	0.410	22.604	10.553	10.455	0.644	9.170	11.350
7	0.433	24.281	9.726	9.652	0.357	8.721	10.031
8	0.358	21.751	11.709	11.698	0.080	11.586	11.837
SSE vectorization							
1	0.766	15.471	5.474	5.478	0.022	5.450	5.529
2	0.404	12.989	10.407	10.333	0.203	10.042	10.563
3	0.352	14.693	11.910	11.789	0.411	10.681	12.105
4	0.247	13.279	17.009	17.005	0.255	16.512	17.332
5	0.354	17.274	11.920	12.541	1.161	11.216	14.265
6	0.314	16.339	13.524	13.340	0.700	11.641	14.097
7	0.318	16.631	13.202	13.339	1.194	11.526	15.151
8	0.242	13.859	17.395	17.380	0.116	17.155	17.524
AVX vectorization							
1	0.743	15.846	5.665	5.678	0.056	5.622	5.794
2	0.391	12.777	10.747	10.718	0.123	10.538	10.871
3	0.376	16.675	11.278	11.188	0.163	10.941	11.377
4	0.242	13.403	17.375	17.425	0.285	17.026	17.992
5	0.339	16.959	12.525	12.602	0.961	11.161	14.527
6	0.305	16.057	14.387	14.073	1.110	11.837	15.935
7	0.302	16.503	14.212	13.557	1.355	11.283	14.881
8	0.236	13.734	17.792	17.756	0.177	17.393	18.002

Table B.7: FFTW performance and energy, N=2^{25}, default scheduling

NT	RT	Energy (Joule)	GFLOPS (median)	GFLOPS (mean)	Standard deviation	GFLOPS (min)	GFLOPS (max)
No vectorization							
1	0.001	0.021	5.211	5.195	0.030	5.127	5.219
2	6e-04	0.018	9.109	9.064	0.091	8.865	9.121
3	6e-04	0.022	9.452	9.445	0.035	9.391	9.514
4	3e-04	0.018	14.815	14.799	0.100	14.623	14.948
5	4e-04	0.020	12.880	12.890	0.126	12.673	13.076
6	4e-04	0.020	13.382	13.400	0.083	13.294	13.542
7	4e-04	0.022	12.558	12.522	0.141	12.261	12.706
8	3e-04	0.020	14.857	14.826	0.095	14.595	14.947
SSE optimizations							
1	4e-04	0.008	13.383	13.386	0.115	13.147	13.520
2	3e-04	0.008	18.560	18.560	0.227	18.241	18.989
3	2e-04	0.009	22.004	22.009	0.179	21.744	22.253
4	2e-04	0.009	27.526	27.576	0.206	27.290	27.955
5	3e-04	0.013	18.105	18.167	0.353	17.718	18.789
6	2e-04	0.010	25.678	25.602	0.271	25.136	26.057
7	3e-04	0.012	20.515	20.294	0.522	19.019	20.744
8	2e-04	0.009	30.561	30.393	0.523	29.153	30.845
AVX optimizations							
1	3e-04	0.007	16.519	16.532	0.116	16.292	16.713
2	2e-04	0.007	23.039	23.056	0.343	22.701	23.650
3	3e-04	0.010	18.577	18.557	0.101	18.337	18.677
4	2e-04	0.008	31.310	31.479	0.819	30.628	33.135
5	3e-04	0.012	19.340	19.286	0.220	18.911	19.536
6	3e-04	0.013	19.022	19.032	0.157	18.719	19.327
7	3e-04	0.013	20.211	20.131	0.252	19.660	20.486
8	2e-04	0.009	31.898	31.875	0.213	31.551	32.259

Table B.8: FFTW performance and energy, N=2^{14}, default scheduling

No vectorization						
N	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)
	1 thread		4 threads		8 threads	
2	0.254	0.0	0.244	0.0	0.245	0.0
4	0.935	0.0	0.936	0.0	0.900	0.0
8	2.191	0.0	2.237	0.0	2.258	0.0
16	4.098	0.0	4.090	0.0	3.889	0.0
32	5.286	0.0	5.220	1e-05	5.050	1e-05
64	6.114	1e-05	6.071	1e-05	5.655	1e-05
128	5.975	2e-05	0.258	6e-04	0.105	0.002
256	6.169	4e-05	0.561	6e-04	0.184	0.002
512	5.761	9e-05	1.120	7e-04	0.405	0.003
1024	5.937	2e-04	1.981	9e-04	0.779	0.003
2048	5.827	4e-04	3.166	0.001	2.404	0.002
4096	5.507	9e-04	4.816	0.002	4.662	0.003
8192	5.437	0.002	5.738	0.004	6.536	0.004
16384	5.259	0.005	8.360	0.006	8.335	0.007
32768	5.096	0.010	11.746	0.010	10.881	0.013
65536	4.886	0.023	14.015	0.020	14.121	0.023
131072	4.909	0.049	16.352	0.039	16.953	0.040
262144	4.790	0.103	17.195	0.080	16.464	0.089
524288	3.363	0.306	12.289	0.234	12.846	0.249
1048576	3.292	0.657	11.248	0.525	11.771	0.554
2097152	3.275	1.398	11.299	1.107	11.807	1.157
4194304	3.279	2.931	11.554	2.303	12.060	2.413
8388608	3.328	6.039	11.384	4.904	11.765	5.163
16777216	3.335	12.480	11.348	10.334	11.468	11.063
33554432	3.259	26.895	11.043	21.520	11.365	22.858

Table B.9: FFTW performance and energy, default scheduling, no vectorization

SSE vectorization						
N	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)
	1 thread		4 threads		8 threads	
2	0.253	0.0	0.251	0.0	0.223	0.0
4	0.901	0.0	0.894	0.0	0.858	0.0
8	2.301	0.0	2.280	0.0	2.096	0.0
16	4.556	0.0	4.515	0.0	4.298	0.0
32	9.458	0.0	6.614	0.0	6.272	0.0
64	13.608	0.0	8.362	1e-05	5.582	1e-05
128	17.434	1e-05	9.283	2e-05	8.696	2e-05
256	17.843	1e-05	1.282	3e-04	0.435	0.001
512	20.936	2e-05	1.261	6e-04	0.992	0.001
1024	20.348	5e-05	2.365	7e-04	0.851	0.002
2048	19.965	1e-04	5.005	8e-04	1.585	0.003
4096	18.554	3e-04	7.527	0.001	4.579	0.002
8192	14.975	8e-04	11.109	0.002	10.526	0.002
16384	14.149	0.002	11.867	0.004	16.003	0.004
32768	12.988	0.004	20.138	0.005	20.923	0.006
65536	11.912	0.009	25.960	0.009	29.474	0.009
131072	12.661	0.018	32.821	0.017	37.526	0.017
262144	12.165	0.041	36.904	0.035	36.930	0.037
524288	5.519	0.181	19.637	0.139	20.902	0.144
1048576	5.392	0.394	17.308	0.329	18.282	0.335
2097152	5.456	0.814	16.995	0.694	17.475	0.729
4194304	5.519	1.703	16.421	1.533	16.974	1.577
8388608	5.612	3.498	16.501	3.168	16.538	3.356
16777216	5.388	7.575	17.105	6.459	17.163	6.832
33554432	5.331	16.191	16.361	13.750	16.794	14.413

Table B.10: FFTW performance and energy, default scheduling, SSE optimized

	AVX vectorization					
N	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)
	1 thread		4 threads		8 threads	
2	0.257	0.0	0.253	0.0	0.244	0.0
4	0.904	0.0	0.900	0.0	0.863	0.0
8	2.395	0.0	2.255	0.0	2.125	0.0
16	4.006	0.0	3.949	0.0	3.773	0.0
32	6.681	0.0	5.301	1e-05	5.061	1e-05
64	15.750	0.0	5.921	1e-05	5.613	1e-05
128	21.957	0.0	0.532	3e-04	0.232	8e-04
256	27.542	1e-05	0.567	6e-04	0.384	0.001
512	30.896	2e-05	1.268	6e-04	0.964	0.001
1024	30.425	4e-05	2.606	7e-04	0.795	0.003
2048	27.802	9e-05	4.843	9e-04	1.647	0.003
4096	24.563	2e-04	8.399	0.001	4.472	0.003
8192	21.036	6e-04	12.902	0.002	10.838	0.002
16384	19.224	0.001	17.781	0.003	18.677	0.003
32768	17.571	0.003	26.653	0.004	27.205	0.005
65536	16.705	0.007	30.926	0.008	32.464	0.009
131072	16.709	0.015	29.658	0.018	44.852	0.015
262144	16.395	0.032	48.895	0.029	49.018	0.030
524288	6.063	0.167	20.239	0.137	22.642	0.136
1048576	5.866	0.364	18.275	0.317	19.247	0.324
2097152	5.895	0.763	17.733	0.685	17.959	0.724
4194304	5.972	1.585	17.761	1.441	17.692	1.559
8388608	6.066	3.262	17.459	3.063	17.928	3.129
16777216	5.832	7.068	17.820	6.359	17.297	6.830
33554432	5.727	15.257	17.434	13.359	17.717	13.800

Table B.11: FFTW performance and energy, default scheduling, AVX optimized

	AVX vectorization					
N	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)	GFLOPS (median)	Energy (Joule)
	1 thread		4 threads		8 threads	
128	11.512	0.008	18.387	0.013	15.506	0.016
256	26.416	0.029	41.570	0.051	22.645	0.098
512	37.988	0.173	97.429	0.186	44.674	0.422
1024	38.092	1.380	137.919	1.129	89.614	1.753
2048	38.703	10.876	138.927	9.019	107.773	11.931
4096	38.997	87.143	145.579	70.305	112.032	92.902
8192	39.096	698.125	149.213	553.921	114.667	734.739

Table B.12: Matrix multiplication performance and energy, default scheduling, AVX optimized

	Black-Scholes		FFTW		Matrix multiplication	
Problem size	**Memory use**	**Accesses per flop**	**Memory use**	**Accesses per flop**	**Memory use**	**Accesses per flop**
2^{1}	0.05 KB	0.0	0.03 KB	0.0	-	-
2^{2}	0.11 KB	0.0	0.06 KB	0.0	-	-
2^{3}	0.22 KB	0.0	0.13 KB	0.0	-	-
2^{4}	0.44 KB	0.0	0.25 KB	0.0	-	-
2^{5}	0.88 KB	0.0	0.5 KB	0.0	-	-
2^{6}	1.75 KB	0.0	1.0 KB	0.0	-	-
2^{7}	3.5 KB	0.0	2.0 KB	0.0	192.0 KB	0.0
2^{8}	7.0 KB	0.0	4.0 KB	0.0	768.0 KB	0.0
2^{9}	14.0 KB	0.0	8.0 KB	0.0	3.0 MB	2e-06
2^{10}	28.0 KB	0.0	16.0 KB	0.0	12.0 MB	3e-05
2^{11}	56.0 KB	0.0	32.0 KB	0.0	48.0 MB	3.8e-05
2^{12}	112.0 KB	0.0	64.0 KB	0.0	192.0 MB	3.8e-05
2^{13}	224.0 KB	0.0	128.0 KB	0.0	768.0 MB	3.6e-05
2^{14}	448.0 KB	0.0	256.0 KB	0.0	-	-
2^{15}	896.0 KB	0.0	512.0 KB	0.0	-	-
2^{16}	1.75 MB	0.0	1.0 MB	0.0	-	-
2^{17}	3.5 MB	1e-06	2.0 MB	2e-06	-	-
2^{18}	7.0 MB	4.7e-05	4.0 MB	7.3e-05	-	-
2^{19}	14.0 MB	0.000123	8.0 MB	0.000494	-	-
2^{20}	28.0 MB	0.000145	16.0 MB	0.002005	-	-
2^{21}	56.0 MB	0.000115	32.0 MB	0.00285	-	-
2^{22}	112.0 MB	9.2e-05	64.0 MB	0.002606	-	-
2^{23}	224.0 MB	8.3e-05	128.0 MB	0.002279	-	-
2^{24}	448.0 MB	8e-05	256.0 MB	0.003502	-	-
2^{25}	896.0 MB	7.2e-05	512.0 MB	0.002144	-	-

Table B.13: Memory accesses per flop

Appendix C

AVX Enabled Logarithms and Exponential Functions

```c
/*
    BEGIN MODIFICATIONS
    AVX additions made by Hallgeir Lien (hallgeir.lien@gmail.com)
*/
#ifdef USE_AVX
#include <immintrin.h>
#include "icc_gcc_compat.h"

typedef __m256 v8sf;
typedef __m256i v8si;

/* natural logarithm computed for 8 simultaneous float
   return NaN for x <= 0
*/
v8sf log256_ps(v8sf x) {
  v8si emm0;
  v8sf one = *(v8sf*)_256ps_1;

  v8sf invalid_mask = _mm256_cmple_ps(x, _mm256_setzero_ps());
  x = _mm256_max_ps(x, *(v8sf*)_256ps_min_norm_pos);  /* cut off denormalized stuff */
  //256 bit shift is not implemented yet; do two 128 bit shifts
  {
      v4si emm01 = _mm_srli_epi32(_mm_castps_si128(_mm256_extractf128_ps(x, 0)), 23);
      v4si emm02 = _mm_srli_epi32(_mm_castps_si128(_mm256_extractf128_ps(x, 1)), 23);
      //256 bit arithmetic not implemented... do it separately
      emm01 = _mm_sub_epi32(emm01, *(v4si*)_pi32_0x7f);
      emm02 = _mm_sub_epi32(emm02, *(v4si*)_pi32_0x7f);

      emm0 = _mm256_insertf128_si256(emm0, emm01, 0);
      emm0 = _mm256_insertf128_si256(emm0, emm02, 1);
  }

  /* keep only the fractional part */
  x = _mm256_and_ps(x, *(v8sf*)_256ps_inv_mant_mask);
  x = _mm256_or_ps(x, *(v8sf*)_256ps_0p5);
  v8sf e = _mm256_cvtepi32_ps(emm0);

  e = _mm256_add_ps(e, one);
  /* part2:
     if( x < SQRTHF ) {
      e -= 1;
      x = x + x - 1.0;
    } else { x = x - 1.0; }
  */
  v8sf mask = _mm256_cmplt_ps(x, *(v8sf*)_256ps_cephes_SQRTHF);
  v8sf tmp = _mm256_and_ps(x, mask);
  x = _mm256_sub_ps(x, one);
  e = _mm256_sub_ps(e, _mm256_and_ps(one, mask));
  x = _mm256_add_ps(x, tmp);


  v8sf z = _mm256_mul_ps(x,x);

  v8sf y = *(v8sf*)_256ps_cephes_log_p0;
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p1);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p2);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p3);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p4);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p5);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p6);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p7);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_log_p8);
  y = _mm256_mul_ps(y, x);

  y = _mm256_mul_ps(y, z);


  tmp = _mm256_mul_ps(e, *(v8sf*)_256ps_cephes_log_q1);
  y = _mm256_add_ps(y, tmp);


  tmp = _mm256_mul_ps(z, *(v8sf*)_256ps_0p5);
  y = _mm256_sub_ps(y, tmp);

  tmp = _mm256_mul_ps(e, *(v8sf*)_256ps_cephes_log_q2);
  x = _mm256_add_ps(x, y);
  x = _mm256_add_ps(x, tmp);
  x = _mm256_or_ps(x, invalid_mask); // negative arg will be NAN
  return x;
}

v8sf exp256_ps(v8sf x) {
  v8sf tmp = _mm256_setzero_ps(), fx;
  v8si emm0 = _mm256_setzero_si256();
  v8sf one = *(v8sf*)_256ps_1;

  x = _mm256_min_ps(x, *(v8sf*)_256ps_exp_hi);
  x = _mm256_max_ps(x, *(v8sf*)_256ps_exp_lo);

  /* express exp(x) as exp(g + n*log(2)) */
  fx = _mm256_mul_ps(x, *(v8sf*)_256ps_cephes_LOG2EF);
  fx = _mm256_add_ps(fx, *(v8sf*)_256ps_0p5);

  /* how to perform a floorf with SSE: just below */
  emm0 = _mm256_cvttps_epi32(fx);
  tmp  = _mm256_cvtepi32_ps(emm0);
  /* if greater, substract 1 */
  v8sf mask = _mm256_cmpgt_ps(tmp, fx);
  mask = _mm256_and_ps(mask, one);
  fx = _mm256_sub_ps(tmp, mask);

  tmp = _mm256_mul_ps(fx, *(v8sf*)_256ps_cephes_exp_C1);
  v8sf z = _mm256_mul_ps(fx, *(v8sf*)_256ps_cephes_exp_C2);
  x = _mm256_sub_ps(x, tmp);
  x = _mm256_sub_ps(x, z);

  z = _mm256_mul_ps(x,x);

  v8sf y = *(v8sf*)_256ps_cephes_exp_p0;
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_exp_p1);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_exp_p2);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_exp_p3);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_exp_p4);
  y = _mm256_mul_ps(y, x);
  y = _mm256_add_ps(y, *(v8sf*)_256ps_cephes_exp_p5);
  y = _mm256_mul_ps(y, z);
  y = _mm256_add_ps(y, x);
  y = _mm256_add_ps(y, one);

  /* build 2^n */
  emm0 = _mm256_cvttps_epi32(fx);
  {
      v4si emm01 = _mm_slli_epi32(_mm_add_epi32(_mm256_extractf128_si256(emm0, 0), *(
    v4si*)_pi32_0x7f), 23),
           emm02 = _mm_slli_epi32(_mm_add_epi32(_mm256_extractf128_si256(emm0, 1), *(
    v4si*)_pi32_0x7f), 23);

      emm0 = _mm256_insertf128_si256(emm0, emm01, 0);
      emm0 = _mm256_insertf128_si256(emm0, emm02, 1);
  }
  v8sf pow2n = _mm256_castsi256_ps(emm0);
  y = _mm256_mul_ps(y, pow2n);

  return y;
}
#endif
/*
    END MODIFICATIONS
*/
```

Appendix D

NEON Enabled Logarithms and Exponential Functions

```c
static inline float32x4_t log128_neon_intrin(float32x4_t x)
{
    //compute invalid mask
    uint32x4_t invalid_mask = vcltq_f32(x, vdupq_n_f32(0.f)),
               tmp2, mask;
    int32x4_t  tmp1;

    float32x4_t e, y, z, tmp1f;

    //cut off denormalized stuff
    x = vmaxq_f32(x, vreinterpretq_f32_u32(vdupq_n_u32(0x00800000)));

    //Shift left by 23 bits
    tmp1 = vshrq_n_s32(vreinterpretq_s32_f32(x), 23);

    //Bitwise-AND with inverse mantissa mask
    //(Keep only fractional part)
    tmp2 = vandq_u32(vreinterpretq_u32_f32(x), vdupq_n_u32(0x807fffff));
    tmp2 = vorrq_u32(tmp2, vreinterpretq_u32_f32(vdupq_n_f32(0.5f)));
    x = vreinterpretq_f32_u32(tmp2);

    tmp1 = vsubq_s32(tmp1, vdupq_n_s32(0x7f));
    e = vcvtq_f32_s32(tmp1);

    e = vaddq_f32(e, vdupq_n_f32(1.0f));

    mask = vcltq_f32(x, vdupq_n_f32(0.707106781186547524f));
    tmp2 = vandq_u32(vreinterpretq_u32_f32(x), mask);
    x = vsubq_f32(x, vdupq_n_f32(1.0f));
    e = vsubq_f32(e, vreinterpretq_f32_u32(vandq_u32(vreinterpretq_u32_f32(vdupq_n_f32
    (1.0f)), mask)));
    x = vaddq_f32(x, vreinterpretq_f32_u32(tmp2));

    z = vmulq_f32(x,x);

    y = vdupq_n_f32(7.0376836292E-2);   //p0
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(-1.1514610310E-1)); //p1
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(1.1676998740E-1));  //p2
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(-1.2420140846E-1)); //p3
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(1.4249322787E-1));  //p4
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(-1.6668057665E-1)); //p5
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(2.0000714765E-1));  //p6
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(-2.4999993993E-1)); //p7
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(3.3333331174E-1)); //p8
    y = vmulq_f32(y, x);

    y = vmulq_f32(y, z);

    tmp1f = vmulq_f32(e, vdupq_n_f32(-2.12194440e-4));
    y = vaddq_f32(y, tmp1f);

    tmp1f = vmulq_f32(z, vdupq_n_f32(0.5f));
    y = vsubq_f32(y, tmp1f);

    tmp1f = vmulq_f32(e, vdupq_n_f32(0.693359375));
    x = vaddq_f32(x, y);
    x = vaddq_f32(x, tmp1f);
    x = vreinterpretq_f32_u32(vorrq_u32(vreinterpretq_u32_f32(x), invalid_mask));

    return x;
}

static inline float32x4_t exp128_neon_intrin(float32x4_t x)
{
    float32x4_t fx, tmp1f, y, z;
    uint32x4_t mask;
    int32x4_t tmp1i;

    x = vminq_f32(x, vdupq_n_f32(88.3762626647949f));
    x = vmaxq_f32(x, vdupq_n_f32(-88.3762626647949f));
    fx = vmulq_f32(x, vdupq_n_f32(1.44269504088896341));
    fx = vaddq_f32(fx, vdupq_n_f32(0.5));

    tmp1f = vcvtq_f32_s32(vcvtq_s32_f32(fx));
    mask = vcgtq_f32(tmp1f, fx);
    mask = vandq_u32(mask, vreinterpretq_u32_f32(vdupq_n_f32(1.0f)));

    fx = vsubq_f32(tmp1f, vreinterpretq_f32_u32(mask));

    tmp1f = vmulq_f32(fx, vdupq_n_f32(0.693359375));
    z = vmulq_f32(fx, vdupq_n_f32(-2.12194440e-4f));
    x = vsubq_f32(x, tmp1f);
    x = vsubq_f32(x, z);

    z = vmulq_f32(x,x);

    y = vdupq_n_f32(1.9875691500E-4f); //p0
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(1.3981999507E-3f));//p1
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(8.3334519073E-3f));//p2
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(4.1665795894E-2f));//p3
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(1.6666665459E-1f));//p4
    y = vmulq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(5.0000001201E-1f));//p5
    y = vmulq_f32(y, z);
    y = vaddq_f32(y, x);
    y = vaddq_f32(y, vdupq_n_f32(1.0f));

    tmp1i = vcvtq_s32_f32(fx);
    tmp1i = vaddq_s32(tmp1i, vdupq_n_s32(0x7f));
    tmp1i = vshlq_n_s32(tmp1i, 23);

    tmp1f = vreinterpretq_f32_s32(tmp1i);
    x = vmulq_f32(y, tmp1f);

    return x;
}
```

Appendix E

SSE, AVX and NEON Accellerated Black-Scholes - Source code

This appendix contains the source code for the relevant parts of the vectorized and OmpSs-parallelized implementation of Black-Scholes.

Listing E.1 Outer loop in Black-Scholes

```
for (ii = 0; ii < array_size; ii+=local_work_group_size)
{
    #pragma omp task private (i)
    {
        for (i=ii; (i<ii+local_work_group_size) && (i<array_size); i+=vector_width)
        {
            #if defined(AVX)
            bsop_avx(&answer_fptr[i], &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &r_fptr
    [i], &sigma_fptr[i], &T_fptr[i]);
            _mm256_zeroupper();
            #elif defined(SSE)
            bsop_sse(&answer_fptr[i], &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &r_fptr
    [i], &sigma_fptr[i], &T_fptr[i]);
            #elif defined(NEON_INTRIN)
            bsop_intrin(&answer_fptr[i], &cpflag_fptr[i], &S0_fptr[i], &K_fptr[i], &
    r_fptr[i], &sigma_fptr[i], &T_fptr[i]);
            #else
            answer_fptr[i] = bsop_reference_float(cpflag_fptr[i], S0_fptr[i], K_fptr[i
    ], r_fptr[i], sigma_fptr[i], T_fptr[i]);
            #endif
        }
    }
}
```

Listing E.2 Black-Scholes formula evaluation, SSE

```
inline void bsop_sse(float* answer, unsigned int* cpflag, float* S0, float* K, float* r, float* sigma, float* T) {
    __m128 _d1, _d2, _c, _p, _Nd1, _Nd2, _expval, _answer, _tmp1, _T, _sigma, _K, _r, _S0;

    __m128 _pt5,
           _one,
           _zero;
        _pt5 = _mm_set1_ps(.5);
        _one  = _mm_set1_ps(1),
        _zero = _mm_set1_ps(0);

    //Loads
    _T       = _mm_loadu_ps(T);
    _sigma   = _mm_loadu_ps(sigma);
    _K       = _mm_loadu_ps(K);
    _r       = _mm_loadu_ps(r);
    _S0      = _mm_loadu_ps(S0);

    // d1 = logf(S0/K)
    _d1      = _mm_div_ps(_S0, _K);
    _d1      = log_ps(_d1);

    // expval = exp(-r*T)
    _expval = _mm_mul_ps(_T, _r);
    //Negate value of r by reversing the sign bit
    __m128 _absmask = _mm_castsi128_ps(_mm_set1_epi32(0x80000000));
    _expval = _mm_xor_ps(_absmask, _expval);
    _expval = exp_ps(_expval);

    // d1 = logf(S0/K) + (r + 0.5*sigma*sigma)*T;
    _tmp1 = _mm_mul_ps(_sigma, _sigma);           // sigma*sigma
    _tmp1 = _mm_mul_ps(_tmp1, _pt5); // 0.5*sigma*sigma
    _tmp1 = _mm_add_ps(_tmp1, _r);                // r + 0.5*sigma*sigma
    _tmp1 = _mm_mul_ps(_tmp1, _T);                // (r + 0.5*sigma*sigma)*T
    _d1    = _mm_add_ps(_d1, _tmp1);              // logf(S0/K) + (r + 0.5*sigma*sigma)*T


    // d1 /= (sigma * sqrt(T));
    _d1    = _mm_div_ps(_d1, _sigma);    // d1 /= sigma
    _d1    = _mm_div_ps(_d1, _mm_sqrt_ps(_T));      // d1 /= (sigma * sqrt(T))

    // d2 = d1 - sigma * sqrt(T);
    _d2    = _mm_sub_ps(_d1, _mm_mul_ps(_sigma, _mm_sqrt_ps(_T)));

    _Nd1 = Nf_sse(_d1);
    _Nd2 = Nf_sse(_d2);

    // c = S0 * Nd1 - K * expval * Nd2;
    _c = _mm_sub_ps(_mm_mul_ps(_S0, _Nd1), _mm_mul_ps(_mm_mul_ps(_K, _expval), _Nd2));

    // p = K * expval * (1.0 - Nd2) - S0 * (1.0 - Nd1);
    _p = _mm_sub_ps(_mm_mul_ps(_K, _mm_mul_ps(_expval, _mm_sub_ps(_one, _Nd2))), // K * expval * (1.0 - Nd2)
                    _mm_mul_ps(_S0, _mm_sub_ps(_one, _Nd1))); // S0 * (1.0 - Nd1)

    _tmp1 = _mm_cmpeq_ps(_mm_loadu_ps((float*)cpflag), _mm_setzero_ps()); // cpflag ?
    _answer = _mm_or_ps(_mm_and_ps(_tmp1, _p), _mm_andnot_ps(_tmp1, _c));
    _mm_storeu_ps(answer, _answer);
}
```

Listing E.3 Standard normal distribution CDF evaluation, SSE

```
inline __m128 Nf_sse(__m128 _x)
{
    __m128 _k, _n, _accum, _candidate_answer, _flag,
           _A1 = _mm_set1_ps(0.319381530),
           _A2 = _mm_set1_ps(-0.356563782),
           _A3 = _mm_set1_ps(1.781477937),
           _A4 = _mm_set1_ps(-1.821255978),
           _A5 = _mm_set1_ps(1.330274429),
           _INV_ROOT2PI = _mm_set1_ps(0.39894228),
           _zero = _mm_set1_ps(0),
           _one = _mm_set1_ps(1),
           _pt5 = _mm_set1_ps(-.5);

    //Get signs of _x
    _flag = _mm_cmplt_ps(_x, _zero);

    //Get absolute value of x by un-setting the sign bit
    __m128 _absmask = _mm_castsi128_ps(_mm_set1_epi32(0x80000000));
    _x = _mm_andnot_ps(_absmask, _x);

    // k = 1.0 / (1.0 + 0.2316419 * x);
    _k = _mm_div_ps(_one, _mm_add_ps(_one, _mm_mul_ps(_mm_set1_ps(0.2316419), _x)));

    _accum = _mm_add_ps(_A4, _mm_mul_ps(_A5, _k));
    _accum = _mm_add_ps(_A3, _mm_mul_ps(_accum, _k));
    _accum = _mm_add_ps(_A2, _mm_mul_ps(_accum, _k));
    _accum = _mm_add_ps(_A1, _mm_mul_ps(_accum, _k));
    _accum = _mm_mul_ps(_accum, _k);

    // n = expf(-0.5 * x * x);
    // n *= INV_ROOT2PI;
    _n = _mm_mul_ps(exp_ps(_mm_mul_ps(_mm_mul_ps(_pt5, _x), _x)), _INV_ROOT2PI);

    // candidate_answer = 1.0 - n * accum;
    _candidate_answer = _mm_sub_ps(_one, _mm_mul_ps(_n, _accum));

    // return (flag ? 1.0 - candidate_answer : candidate_answer);
    _candidate_answer = _mm_or_ps(_mm_andnot_ps(_flag, _candidate_answer),
                              _mm_and_ps(_flag, _mm_sub_ps(_one, _candidate_answer)
    ));

    return _candidate_answer;
}
```

Listing E.4 Black-Scholes formula evaluation, AVX

```
inline void bsop_avx(float* answer, unsigned int* cpflag, float* S0, float* K, float* r, float* sigma, float* T) {
    __m256 _d1, _d2, _c, _p, _Nd1, _Nd2, _expval, _answer, _tmp1, _T, _sigma, _K, _r, _S0;

    __m256 _pt5,
           _one,
           _zero;
        _pt5 = _mm256_set1_ps(.5);
        _one  = _mm256_set1_ps(1),
        _zero = _mm256_set1_ps(0);

    //Loads
    _T      = _mm256_loadu_ps(T);
    _sigma  = _mm256_loadu_ps(sigma);
    _K      = _mm256_loadu_ps(K);
    _r      = _mm256_loadu_ps(r);
    _S0     = _mm256_loadu_ps(S0);

    // d1 = logf(S0/K)
    _d1     = _mm256_div_ps(_S0, _K);
    _d1     = log256_ps(_d1);

    // expval = exp(-r*T)
    _expval = _mm256_mul_ps(_T, _r);
    // Negate value of r by reversing the sign bit
    __m256 _absmask = _mm256_castsi256_ps(_mm256_set1_epi32(0x80000000));
    _expval = _mm256_xor_ps(_absmask, _expval);
    _expval = exp256_ps(_expval);

    // d1 = logf(S0/K) + (r + 0.5*sigma*sigma)*T;
    _tmp1 = _mm256_mul_ps(_sigma, _sigma);           // sigma*sigma
    _tmp1 = _mm256_mul_ps(_tmp1, _pt5); // 0.5*sigma*sigma
    _tmp1 = _mm256_add_ps(_tmp1, _r);                // r + 0.5*sigma*sigma
    _tmp1 = _mm256_mul_ps(_tmp1, _T);                // (r + 0.5*sigma*sigma)*T
    _d1   = _mm256_add_ps(_d1, _tmp1);               // logf(S0/K) + (r + 0.5*sigma*sigma)*T


    // d1 /= (sigma * sqrt(T));
    _d1   = _mm256_div_ps(_d1, _sigma);   // d1 /= sigma
    _d1   = _mm256_div_ps(_d1, _mm256_sqrt_ps(_T));      // d1 /= (sigma * sqrt(T))

    // d2 = d1 - sigma * sqrt(T);
    _d2   = _mm256_sub_ps(_d1, _mm256_mul_ps(_sigma, _mm256_sqrt_ps(_T)));

    _Nd1 = Nf_avx(_d1);
    _Nd2 = Nf_avx(_d2);

    // c = S0 * Nd1 - K * expval * Nd2;
    _c = _mm256_sub_ps(_mm256_mul_ps(_S0, _Nd1), _mm256_mul_ps(_mm256_mul_ps(_K, _expval), _Nd2));

    // p = K * expval * (1.0 - Nd2) - S0 * (1.0 - Nd1);
    _p = _mm256_sub_ps(_mm256_mul_ps(_K, _mm256_mul_ps(_expval, _mm256_sub_ps(_one, _Nd2))), // K * expval * (1.0 - Nd2)
                    _mm256_mul_ps(_S0, _mm256_sub_ps(_one, _Nd1))); // S0 * (1.0 - Nd1)

    _tmp1 = _mm256_cmp_ps(_mm256_loadu_ps((float*)cpflag), _zero, 0); // cpflag ?
    _answer = _mm256_or_ps(_mm256_and_ps(_tmp1, _p), _mm256_andnot_ps(_tmp1, _c));
    _mm256_storeu_ps(answer, _answer);
}
```

Listing E.5 Standard normal distribution CDF evaluation, AVX

```
inline __m256 Nf_avx(__m256 _x)
{
    __m256 _k, _n, _accum, _candidate_answer, _flag,
           _A1 = _mm256_set1_ps(0.319381530),
           _A2 = _mm256_set1_ps(-0.356563782),
           _A3 = _mm256_set1_ps(1.781477937),
           _A4 = _mm256_set1_ps(-1.821255978),
           _A5 = _mm256_set1_ps(1.330274429),
           _INV_ROOT2PI = _mm256_set1_ps(0.39894228),
           _zero = _mm256_set1_ps(0),
           _one = _mm256_set1_ps(1),
           _pt5 = _mm256_set1_ps(-.5);

    //Get signs of _x
    _flag = _mm256_cmplt_ps(_x, _zero);

    //Get absolute value of x by un-setting the sign bit
    __m256 _absmask = _mm256_castsi256_ps(_mm256_set1_epi32(0x80000000));
    _x = _mm256_andnot_ps(_absmask, _x);

    // k = 1.0 / (1.0 + 0.2316419 * x);
    _k = _mm256_div_ps(_one, _mm256_add_ps(_one, _mm256_mul_ps(_mm256_set1_ps
     (0.2316419), _x)));

    _accum = _mm256_add_ps(_A4, _mm256_mul_ps(_A5, _k));
    _accum = _mm256_add_ps(_A3, _mm256_mul_ps(_accum, _k));
    _accum = _mm256_add_ps(_A2, _mm256_mul_ps(_accum, _k));
    _accum = _mm256_add_ps(_A1, _mm256_mul_ps(_accum, _k));
    _accum = _mm256_mul_ps(_accum, _k);

    // n = expf(-0.5 * x * x);
    // n *= INV_ROOT2PI;
    _n = _mm256_mul_ps(exp256_ps(_mm256_mul_ps(_mm256_mul_ps(_pt5, _x), _x)),
    _INV_ROOT2PI);

    // candidate_answer = 1.0 - n * accum;
    _candidate_answer = _mm256_sub_ps(_one, _mm256_mul_ps(_n, _accum));

    // return (flag ? 1.0 - candidate_answer : candidate_answer);
    _candidate_answer = _mm256_or_ps(_mm256_andnot_ps(_flag, _candidate_answer),
                              _mm256_and_ps(_flag, _mm256_sub_ps(_one,
    _candidate_answer)));

    return _candidate_answer;
}
```

Listing E.6 Black-Scholes formula evaluation, NEON

```
void bsop_neon_intrin(float* answer, unsigned int* _cpflag, float* _S0, float* _K,
    float* _r, float* _sigma, float* _T) {
    float32x4_t T, sigma, K, r, S0, Nd1, Nd2, tmp1f, tmp2f;
    uint32x4_t mask; int32x4_t cpflag;
    /* bulk load all arguments */
    T      = vld1q_f32(_T);
    sigma  = vld1q_f32(_sigma);
    K      = vld1q_f32(_K);
    r      = vld1q_f32(_r);
    S0     = vld1q_f32(_S0);
    cpflag = vld1q_s32(_cpflag);

    //find logf(S0/K). Estimate reciprocal of K
    Nd1 = vrecpeq_f32(K);
    Nd1 = vmulq_f32(Nd1, vrecpsq_f32(Nd1, K));
    Nd1 = vmulq_f32(Nd1, vrecpsq_f32(Nd1, K));
    Nd1 = vmulq_f32(Nd1, vrecpsq_f32(Nd1, K));
    Nd1 = vmulq_f32(Nd1, vrecpsq_f32(Nd1, K));

    Nd1 = vmulq_f32(Nd1, S0);
    Nd1 = log128_neon_intrin(Nd1);
    //Nd1 = logf(S0/K) + (r + 0.5*sigma*sigma)*T;
    Nd1 = vaddq_f32(Nd1, vmulq_f32(vaddq_f32(r, vmulq_f32(vdupq_n_f32(0.5), vmulq_f32(
    sigma, sigma))), T));
    //sqrt(T)
    tmp1f = vrsqrteq_f32(T);
    tmp1f = vmulq_f32(tmp1f, vrsqrtsq_f32(vmulq_f32(tmp1f,tmp1f), T));
    tmp1f = vmulq_f32(tmp1f, vrsqrtsq_f32(vmulq_f32(tmp1f,tmp1f), T));
    tmp1f = vmulq_f32(tmp1f, vrsqrtsq_f32(vmulq_f32(tmp1f,tmp1f), T));
    tmp1f = vmulq_f32(tmp1f, vrsqrtsq_f32(vmulq_f32(tmp1f,tmp1f), T));
    //find reciprocal of 1/sqrt(T)
    tmp2f = vrecpeq_f32(tmp1f);
    tmp2f = vmulq_f32(tmp2f, vrecpsq_f32(tmp2f, tmp1f));
    tmp2f = vmulq_f32(tmp2f, vrecpsq_f32(tmp2f, tmp1f));
    tmp2f = vmulq_f32(tmp2f, vrecpsq_f32(tmp2f, tmp1f));
    tmp2f = vmulq_f32(tmp2f, vrecpsq_f32(tmp2f, tmp1f));
    //1/(sigma*sqrt(T))
    Nd2 = vrecpeq_f32(sigma);
    Nd2 = vmulq_f32(Nd2, vrecpsq_f32(Nd2, sigma));
    Nd2 = vmulq_f32(Nd2, vrecpsq_f32(Nd2, sigma));
    Nd2 = vmulq_f32(Nd2, vrecpsq_f32(Nd2, sigma));
    Nd2 = vmulq_f32(Nd2, vrecpsq_f32(Nd2, sigma));
    Nd2 = vmulq_f32(Nd2, tmp1f);

    //d1/(sigma*sqrt(T)), d2=d1-sigma*sqrt(T)
    Nd1 = vmulq_f32(Nd1, Nd2);
    Nd2 = vsubq_f32(Nd1, vmulq_f32(sigma, tmp2f));

    Nd1 = Nf_neon_intrin(Nd1);
    Nd2 = Nf_neon_intrin(Nd2);

    //expval = exp(-r*T)
    tmp2f = vnegq_f32(vmulq_f32(T, r));
    tmp2f = exp128_neon_intrin(tmp2f);

    //use sigma as placeholder for c
    sigma = vsubq_f32(vmulq_f32(S0, Nd1), vmulq_f32(vmulq_f32(K, tmp2f), Nd2));
    //use T as placeholder for p
    T = vsubq_f32(vmulq_f32(K, vmulq_f32(tmp2f, vsubq_f32(vdupq_n_f32(1.0f), Nd2))),
                vmulq_f32(S0, vsubq_f32(vdupq_n_f32(1.0f), Nd1)));

    mask = vceqq_s32(cpflag, vdupq_n_s32(0));
    mask = vorrq_u32(vandq_u32(mask, vreinterpretq_u32_f32(T)), vbicq_u32(
    vreinterpretq_u32_f32(sigma), mask));
    vst1q_f32(answer, vreinterpretq_f32_u32(mask));
}
```

Listing E.7 Standard normal distribution CDF evaluation, NEON

```c
static inline float32x4_t Nf_neon_intrin(float32x4_t x)
{
    uint32x4_t flag;
    float32x4_t k, recp_accum, n;

    //get signs of x
    flag = vcltq_f32(x, vdupq_n_f32(0));

    //get absolute value
    x = vabsq_f32(x);

    // k = 1.0 / (1.0 + 0.2316419 * x);
    k = vmulq_f32(x, vdupq_n_f32(0.2316419));
    k = vaddq_f32(k, vdupq_n_f32(1));

    //find reciprocal
    recp_accum = vrecpeq_f32(k);
    recp_accum = vmulq_f32(recp_accum, vrecpsq_f32(recp_accum, k));
    recp_accum = vmulq_f32(recp_accum, vrecpsq_f32(recp_accum, k));
    recp_accum = vmulq_f32(recp_accum, vrecpsq_f32(recp_accum, k));
    k = vmulq_f32(recp_accum, vrecpsq_f32(recp_accum, k));

    //Accumulation step
    recp_accum = vaddq_f32(vdupq_n_f32(-1.821255978), vmulq_f32(vdupq_n_f32
    (1.330274429), k));
    recp_accum = vaddq_f32(vdupq_n_f32(1.781477937), vmulq_f32(recp_accum, k));
    recp_accum = vaddq_f32(vdupq_n_f32(-0.356563782), vmulq_f32(recp_accum, k));
    recp_accum = vaddq_f32(vdupq_n_f32(0.319381530), vmulq_f32(recp_accum, k));
    recp_accum = vmulq_f32(recp_accum, k);


    n = vmulq_f32(vmulq_f32(vdupq_n_f32(-0.5f), x), x);
    n = vmulq_f32(exp128_neon_intrin(n), vdupq_n_f32(0.39894228f));

    //candidate answer: n
    n = vsubq_f32(vdupq_n_f32(1.0f), vmulq_f32(n, recp_accum));

    n = vreinterpretq_f32_u32(vorrq_u32(vbicq_u32(vreinterpretq_u32_f32(n), flag),
                              vandq_u32(flag, vreinterpretq_u32_f32(vsubq_f32(
    vdupq_n_f32(1.0f), n))))
    );

    return n;
}
```

Appendix F

FFTW Implementation with OmpSs

This appendix contains the source code for the relevant parts of the vectorized and OmpSs-parallelized implementation of FFTW, as well as the benchmark software. Note that command line parsing, some error checking and verbose output was stripped from listing F.2 due to space considerations.

Listing F.1 Relevant lines in threads/openmp.c for FFTW that were ported to OmpSs. Changes are clearly commented with ADDED and REMOVED.

```
THREAD_ON; /* prevent debugging mode from failing under threads */
/* REMOVED #pragma omp parallel for */
for (i = 0; i < nthr; ++i)
{
    /* ADDED task block */
    #pragma omp task private(d)
    {
        d.max = (d.min = i * block_size) + block_size;
        if (d.max > loopmax)
            d.max = loopmax;
        d.thr_num = i;
        d.data = data;
        proc(&d);
    }
}
/* ADDED synchronization */
#pragma omp taskwait
THREAD_OFF; /* prevent debugging mode from failing under threads */
```

Listing F.2 Relevant source code from the FFTW benchmark

```c
int main(int argc, char *argv[]) {
    int i, rc; double t0; cmd_args args;

    //Parse the command line
    parse_cmdline(argc, argv, &args);

    // Create input/output arrays
    int n[3] = {args.width, args.height, args.depth};
    size_t N = args.width*args.height*args.depth, Nbytes = (N<8192?8192:N)*sizeof(
    fftwf_complex);

    // Initialize threading
    fftwf_init_threads();
    fftwf_plan_with_nthreads(args.threads);

    // Set up FFTW
    fftwf_complex* in = 0, * out = 0;
    fftwf_plan p;

    in = (fftwf_complex*) fftwf_malloc(Nbytes);
    if (args.inplace) out = in;
    else out = (fftwf_complex*) fftwf_malloc(Nbytes);

    if (!in || !out) printf("Input array errors: %lld %lld\n", in, out);

    char wisdom_filename[255], * schedule = getenv("NX_SCHEDULE");
    sprintf(wisdom_filename,"wisdom_%s_%s.wiz",vectorize,schedule?schedule:"default");
    fftwf_import_wisdom_from_filename(wisdom_filename);
    p=fftwf_plan_dft(3,n,in,out,FFTW_FORWARD,args.measure?FFTW_MEASURE:FFTW_ESTIMATE);
    fftwf_export_wisdom_to_filename(wisdom_filename);

    fftwf_execute(p); //execute once to avoid cold starts

    int r, events[] = {PAPI_L2_TCA, PAPI_L2_TCM, PAPI_L3_TCM};
    long long values[4] = {0}, niters = 0;
    PAPI_library_init(PAPI_VER_CURRENT);
    PAPI_thread_init((long unsigned int (*)(void))omp_get_thread_num);

    //Start cachegrind/callgrind collection
    if (args.valgrind) CALLGRIND_TOGGLE_COLLECT;
    PAPI_start_counters(events, 3); // Start performance counters
    energy_counter eng;
    initialize(&eng, NULL);
    start_reading(&eng); //start reading energy counter

    // Start timing
    t0 = -gettime();
    while (t0 + gettime() < 1 && !(niters >= 1 && args.valgrind)) {
        fftwf_execute(p); niters++; }
    t0 = (t0 + gettime()) / (double)niters;

    //stop energy counter
    stop_reading(&eng);
    estimate_energy(&eng);
    PAPI_stop_counters(values, 3);
    if (args.valgrind) CALLGRIND_TOGGLE_COLLECT;

    double mflops = 0.000005*N*log2(N)/t0;
    printf("l2_refs: %lld l2_miss: %lld l3_refs: %lld l3_miss: %lld\n", values[0]/
    niters, values[1]/niters, values[1]/niters, values[2]/niters);
    printf("rt:%e mflops:%e energy:%e\n",t0,mflops,eng.package_energy/(double)niters);

    fftwf_destroy_plan(p); fftwf_free(in);
    if (!args.inplace) fftwf_free(out);
    return 0;
}
```

Appendix G

Experiment Framework

To simplify running experiments in a consistent manner, as well as managing, extracting and formatting the results in a easy way, a significant amount of work was put into writing a framework for running experiments, and extracting the relevant results, formatting them into tables and plots. The framework has two parts: The experiment part, where the experiments are run according to an experiment specification, and a presentation part, where the results are extracted, formatted, and plotted, using another set of scripts. Both parts of the framework is written in Python, and uses sqlite for storage of results. This chapter will briefly present this framework for doing experiments.

G.1 Experiment Scripts

The main program on the experiment side is `test.py`. The syntax in its simplest form is

```
./test.py -i <experiment specification file> -o <database file>
         [-ib N] [-l <logfile>] [-v]
```

Here, the `-i` parameter specifies the experiment specification file to execute, whose format is described in section G.1.1, `-o` specifies the database filename for storage of the results. `-ib` controls the order of the experiments; this is also described in section G.1.1. Lastly, `-l` specifies a log file for more verbose output and errors than what is dumped to the terminal by default, and the `-v` switch turns on verbose mode, which gives a much more detailed output, like for instance the raw program outputs.

G.1.1 Experiment specification files and test suites

`test.py` contains the code for parsing a experiment specification file, and then performing the experiments specified in this file. However, every benchmark and program has its own set of input arguments, and its own names for each of the input arguments. Some programs also require some of the arguments to be sent in as environment vari-

Keyword	Description
`test`	Specifies the name of the experiment, e.g. "blackscholes".
`testset`	Names a subset of the current experiment, e.g. "performance_wf_scheduling".
`include`	Includes other specification files into this file.
`path`	Specifies the path of the executable and makefile for the benchmark.
`exec`	Specifies the name of the executable for the benchmark.
`reps`	Specifies the number of repetitions to run the benchmark for.
`env`	Specifies environment variables to be set. Several variables can be set by separating each VAR=VAL by a space.
`makeflags`	Specifies any special flags to pass to make when building the benchmark.
`argv`	Specifies any extra command line arguments to send verbatim to the benchmark.
`cpufreq`	If CPU frequency control is available, specify the clock frequency in MHz. Note that this requires a script `set_cpufreq` in the same directory as test.py that takes in one argument, the CPU frequency in KHz, and sets the CPU frequency according to this.
`run`	Specify that the benchmark should be run with the parameters that have been set so far.

Table G.1: Reserved keywords/variables in the experiment framework

ables. For instance, Nanos++ requires that the number of threads are given by the environment variable `NX_PES`. Because of this, in addition to having a experiment specification file, test.py contains classes which in this script is called "test suites", which describe how the values given by the experiment specification file should be interpreted, and how the results should be extracted. Which test suite that should be used for a given experiment specification file is determined by looking at the executable name in the specification. Section G.1.3 gives an overview of the required member functions for a test suite, and how to integrate new test suites into the script.

The experiment specification files specify input parameters, command line arguments, environment variables, paths, and so on. The file format is a simple line-by-line key-value format, where each line is either a comment (if the line starts with a #), an empty line (no characters other than newline or whitespace), or a line starting with a key (a "variable" to be set), whose value will be set to whatever comes after this key. The semantics of a variable, or if a variable is even used, is up to the test suite for the program being tested.

Some variables are "reserved", in the sense that they imply special meaning to the framework itself. Those variables or keywords are given in table G.1. Any other variables is parsed and put in a dictionary in test.py, and is not used unless the test suite uses it. An example experiment specification file is given in listing G.1.1. In this example, there is two sets of input parameters, one with `width=1024` and one with `width=2048`. The benchmark is run with each of these input parameters ten times.

The default behavior of the experiment script, `test.py`, is to run `reps` times with the first set of input parameters, then `reps` times with the second set of input

Listing G.1 Example experiment specification file

```
include include/ompss_flags.inc

test bsop_debug
path ./bsop
exec bsop.x

reps 10

env NX_SCHEDULE=wf

# Performance testing, with different sizes
testset perf_novec
cpufreq 3400
vectorize none
nthreads 8
width 1024
run
width 2048
run
```

parameters, and so on. Due to disturbances in the test bench from e.g. system processes or other users, it may be desirable to spread out the experiments so that if there is a heavy system process starts interfering for a certain amount of time, it won't pollute all the results for just one single input, which may lead to wrong conclusions, but instead pollute only a few of the results for each input. To determine the execution order, the command line argument `-ib` can be used. Setting `-ib` to 1 results in a breadth-first execution, while not setting it is equivalent to setting it to infinity, which is depth-first execution, which is the default.

G.1.2 Database layout

For storage, sqlite is used, with the following database schema:

```
CREATE TABLE result_set (test_name VARCHAR(200), testset
VARCHAR(200), <input fields>, <result fields>);
CREATE TABLE stats (test_name VARCHAR(200), testset
VARCHAR(200), <input fields>, <stats fields>);
```

The input fields are given in test.py, in the `input_values` field of the `DB` class, as an array of (<field name>, <type>) tuples. The supported types are the Python types int, str, float and datetime. These are simply the input parameters that should be stored with the results. Some typical input parameters that often should have a field in the database is the problem size and number of threads.

The result fields is given by the `result_values` array in the `DB` class which, like `input_values`, is an array of (<field name>, <type>) tuples. These are the results that should be stored to the database. Typically, all results should be stored, however, exactly what values "all results" encompasses varies. For instance, a researcher measuring energy consumption in some application may want a "energy consumed" field, while someone purely interested in performance benchmarks have no use for such results.

The stats fields are derived from the result fields, in the following way. For each result field, append `_median`, `_mean`, `_stddev`, `_range_min` and `_range_max`.These fields contain the statistical data for the results.

In addition to the tables, there exists a view called "results" on top of the tables result_set and stats, that may contain derived values like speedup, as well as all the statistics for each of the tests. This view is unfortunately hard coded and may have to be modified manually, or removed.

G.1.3 Adding support for more benchmarks

Adding support for another application or benchmark in test.py is fairly simple. First, determine if the benchmark has any input parameters or result fields that should be stored in the result set that do not have any database field already, and if so, add these fields as described in section G.1.2. Then, create another test suite class. The simplest is copying one of the classes that are there already. A test suite class must have the member functions given in table G.2.

The function that actually executes the benchmark and reads out the results is `test_common`. `test_common` takes 9 arguments. The first two is the Test instance and Input instance which represents the global test parameters like the test name and input parameters. The third should be a dictionary of result keys (same as the result database field names) to ResultValue instances, which has the constructor `ResultValue(<type>, <regex>)`. The regex is the regular expression matching the particular result value of interest from the output. The type is the same as the type in the database field. For instance, if the running time of an application has the following format: "The running time was 5.38s", a ResultValue for this output could be constructed as `ResultValue(float, "The running time was S+s")`. The fourth and fifth arguments are the argument list and environment variable list constructed with `suite.arglist` and `suite.envlist`. The sixth and seventh are log and error files, and should just be passed on. The eight, `iters_before_switch`, is the `-ib` parameter. The ninth, `niters_regex`, is a regex for the number of iterations that a benchmark went through before exiting. This is necessary to get the correct means and standard deviations when a benchmark runs multiple times on its own, independently from the `reps` variable.

G.2 Results Extraction and Presentation

On the presentation side, the main script is `maketable.py`, which parses an output specification, which is a file containing the filenames and queries for each file or table to be output, and outputs tables and data files based on this specification. These files can then be handled further, by e.g. parsing them in matplotlib, or with Gnuplot, or inserting them as tables in Latex. The syntax of `maketable.py` is

```
./test.py -i <output specification file>
```

The output specification file has a format similar to the experiment specification file in section G.1.1. Lines starting with # is comments, blank lines are ignored, and

Function	Return value and remarks
`must_make(inext, iprev, test)`	Returns True if the benchmark must be rebuilt, based on the previous input parameters `iprev`, and the next input parameters `inext`, and False otherwise. A good rule is often to return True if `iprev` is None, i.e. the next input parameters to be processed is the first.
`make(test, input , logfiles, errorfiles)`	Returns nothing. Builds any extra arguments to Make before building, based on the input parameters. For instance, in Black-Scholes, any vectorization must be specified as a flag to make. Finally this function should call the global function `make`. In its simplest form, if no extra arguments must be constructed, `suite.make` can simply call `make(test.path, test.makeflags, logfiles, errorfiles)`.
`arglist(test, input, executable)`	This function should return the argument list for the program. Note that the program path and executable name is part of this argument list. For a simple benchmark that only has one input argument, "width", could have a rule like this: `return os.path.join(test.path, test.executable) + " --arraysize " + str(int(input.input_params.get("width", 1)))`
`envlist(test, input)`	This function should return a dictionary containing environment variable names as keys, and their value as the value.
`test(test, input , executable , logfiles, errorfiles, iters_before_switch )`	This function should return the value of a call to `test_common`, whose arguments is described below.

Table G.2: Required member functions of test suites

the rest have a key-value format. The file should start with `set db <dbfile>`, where `<dbfile>` is the results database from the experiments. Then for each output table or datafile, there should be a block like the following one, where the optional lines are enclosed in [and].

```
file <filename>
[separator <field separator string>]
[newline <newline string>]
[direction <direction>]
query <SQL query string 1>
[query <SQL query string 2>]
...
[query <SQL query string N>]
```

The filename is the output filename. For instance, if we were to produce a latex table from the results, we might name it `promising_results.tex`. The `separator` field specifies what string that should separate the fields in the output file, if any. For a latex table, this is an ampersand, "&". The `newline` field specifies a string to be appended before a newline. For latex tables, this should be a double backslash, "

". The `direction` field specifies if additional query results should be appended as rows or columns. "down" means appending additional results as extra rows, while "right" makes additional queries append the results as columns. Finally, the `query` fields can be arbitrarily many, and specifies the SQL queries to perform against the results database.

A simple example which produces a latex table containing the running time, rt, of an application for different problem sizes, N, is shown in listing G.2.

Listing G.2 Example output specification file

```
set db results.db

file data/performance.tex
separator &
newline \\
query select N, rt from results where test_name='mybenchmark' and testset='
    my_new_scheduling_alg'
```

Appendix H

Paper

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

Hallgeir Lien[1], Lasse Natvig[1], Abdullah Al Hasib[1], and Jan Christian Meyer[2]

[1] Dept. of Computer and Information Science (IDI), NTNU, Trondheim, NO-7491, Norway. E-mail: Lasse@computer.org
[2] High Performance Computing Section, IT Dept., NTNU

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

Keywords: performance evaluation, energy efficiency, task based programming.

1 Introduction

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

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

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

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

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

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

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

2 Background

2.1 Execution Platform, SSE and AVX

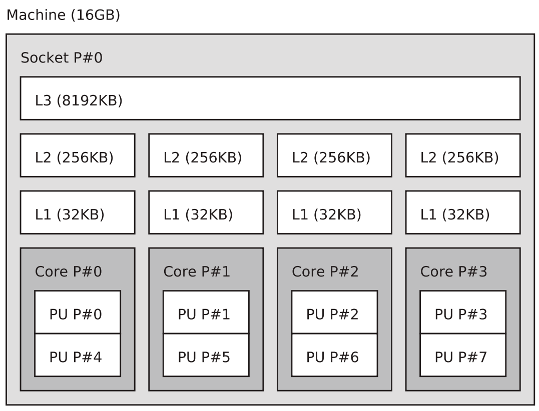
All experiments were executed on a four core desktop computer that can execute 8 threads using Intel Hyperthreading$^{\text{TM}}$. Its main architecture and specifications are shown in Figure 1.

All cores were clocked at their maximum rate of 3.4 GHz. Cache sizes, line sizes and associativity are described in Table 1. Latencies are taken from [7].

The Intel Sandy Bridge processors allow vectorization using SSE or AVX. AVX registers extend the 128 bit SSE registers with an additional 128 bits, and can theoretically double the throughput [8]. SSE and AVX are programmed using intrinsics, inline assembly, or automatic vectorization by the compiler.

2.2 Performance and Energy Metrics

There is a trade-off between the partly conflicting goals of high performance and low energy consumption. Comparing systems based on energy consumption alone



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

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

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

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

would motivate the use of very slow processors with low frequency, since energy is the product of power and execution time. The *Energy-Delay Product* (EDP) places greater emphasis on performance, and corresponds to the reciprocal of performance pr. energy unit.

Different metrics are appropriate to different cases when studying energy efficiency. Rivoire *et al.* [9] give a readable introduction to the pros and cons of various energy efficiency metrics. $Performance^N/Watt$ is among the most general, as it allows adjusting the balance between high performance and low energy consumption. $N = 0$ implies a focus on the power consumption alone, while $N = 2$ corresponds to EDP.

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

Energy measurements were obtained from the energy consumption fields of the non-architectural Machine State Registers (MSRs) made available by the Running Average Power Limit (RAPL) interface [11]. Because these values only reflect chip level energy consumption, we observe the L3 miss rate to find the range of problem sizes where the application is being executed on-chip. As long as the L3 miss rate is close to zero, our on-chip energy measurements give a fair comparison of energy efficiency for the different implementations.

2.3 Selection of Benchmarks

Our choice of applications is motivated by the Mont Blanc project, leading to use of OmpSs, and benchmark selection from potential target applications [5].

Black-Scholes is part of the PARSEC Benchmark Suite for shared memory computers [12]. It calculates prices for a portfolio of European stock options by evaluating the Black-Scholes formula for each element of a data set. A financial market is modeled by repeating this computation over time.

FFTW (Fastest Fourier Transform in the West) is a widely used FFT library. The FFTW library achieves high performance by automatically adapting its algorithm for the machine it is run on. It first creates a plan of execution for the given problem, and then executes it. A plan is created by heuristically tailoring execution to the current system (*e.g.* querying cache sizes), and several different plans are tested to find the fastest candidate. Measurement can be omitted to save plan creation time, when less efficient execution is acceptable [13].

The third application studied is *Matrix Multiplication* implemented with OmpSs. It creates tasks from multiplication tiles, calling BLAS *gemm* at the tile level. We use the ATLAS library for this, because of its AVX support.

3 Experiments and Results

3.1 Method

We use the RAPL MSR interface to read out energy used by the processor chip. The bits 12:8 of the `MSR_RAPL_POWER_UNIT` register describe the granularity of the energy values. The default value is $2^{-16}J \approx 15.3\mu J$. Consumed energy is read from the bits 31:0 of the `MSR_PKG_ENERGY_STATUS` register, which has a wrap-around time of about 60 seconds on high processor load [11]. Our experiments complete in a few seconds, remaining safely within this limit.

Data access was kept within the multi-core chip by limiting problem sizes to fit in the last level cache (LLC). As the RAPL registers do not reflect the cost of off-chip memory, its magnitude is not visible in our results, making it necessary to restrict its influence.

LLC miss rates were recorded using performance monitoring counters, in order to validate that predicted limits for on-chip problem sizes are correct.

The changes in application behavior observed at the LLC limit are visible in our performance results. Every experiment was run 10 times and we plot the median value for each problem size. The first sample points are discarded, in order to remove cache cold start effects.

The results are reproducible and stable, with a relative standard deviation less than 3% for the relevant problem sizes. The standard deviations of runs are far smaller than the margins separating different implementations.

All experiments were run under openSuse 11.4 (x86_64) running Linux kernel 2.6.37.6, and all OmpSs applications were compiled using `sscc` from the OmpSs package. As `sscc` translates at source level, `gcc` 4.7.0 generated the native code. Nanos++ runtime version 0.6a was used for all experiments.

3.2 Black-Scholes

Vectorization of Black-Scholes made it necessary to implement natural logarithm and exponential functions. We adapted code from the Cephes Mathematical Library [14]; further details can be found in H. Lien's Master thesis [10].

Black-Scholes uses 6 input- and one output-array, each containing N 32-bit floating point numbers, giving a memory footprint of $28N$ bytes, where N is the problem size. The largest problem that can fit the LLC is $N = 2^{18}$, as $2^{18} \cdot 28B = 7MB$. The LLC miss rate is below 0.1% for N up to and including 2^{15}, 0.56% for $N = 2^{16}$ were the memory footprint is 1.75 MB, and it increases dramatically for $N = 2^{17}$ and larger problems. Results are shown in Figure 2 and Figure 3.

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

3.3 FFTW

FFTW already supports OpenMP, which allowed us to create a straightforward OmpSs port. This was done by replacing `omp parallel for` constructs with `omp task` loop bodies, and their associated implicit barriers with `omp taskwait`.

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

3.4 Matrix multiplication

Our initial experiments with the OmpSs Matrix Multiply use ATLAS with AVX. They give a peak performance at 149.7 GFLOPS running 4 threads on a

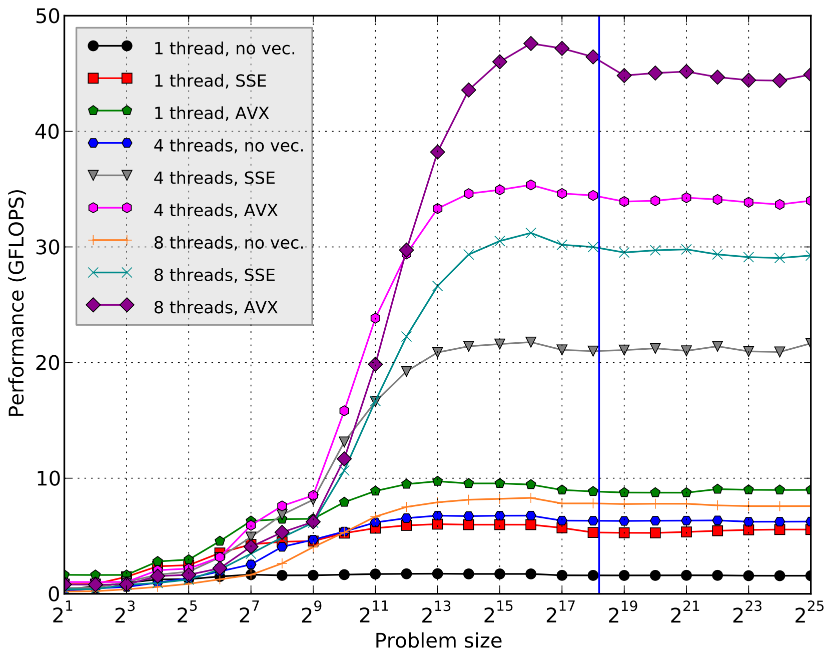


Fig. 2. Performance vs. problem size for Black-Scholes. The vertical line marks the *8MB point*, *i.e.* the problem size where application data require the entire LLC.

8192x8192 matrix. The peak on-chip energy efficiency is 1.97 GFLOPS/W for the same configuration, and we found the LLC misses per floating point operation to be less or equal to $3.8 \cdot 10^{-5}$ for all problem sizes. The results are summarized in Figure 6 and Figure 7.

3.5 Discussion

We compare observations of energy efficiency improvement to corresponding parallel speedup, in order to evaluate the benefit of adding parallelism.

As seen in Figures 2 and 3, Black-Scholes scales favorably. 4-thread runs become advantageous at problem sizes $N = 2^{12}$ and $N = 2^{13}$, and 8-thread runs show energy benefits upwards of $N = 2^{14}$. It is also visible that Black-Scholes retains energy efficiency for out-of-cache problem sizes, albeit with a peak at $N = 2^{16}$. Speedup with hyperthreading (8 threads) is distinctly sub-linear, but there is a clear improvement which admits evaluation of the return on energy investment.

Figures 4 and 5 show that FFTW reaps no benefit from hyperthreading, and clearly becomes bandwidth bound for problem sizes beyond available cache space. This limit is characteristic of the kernel, and also witnessed by the results

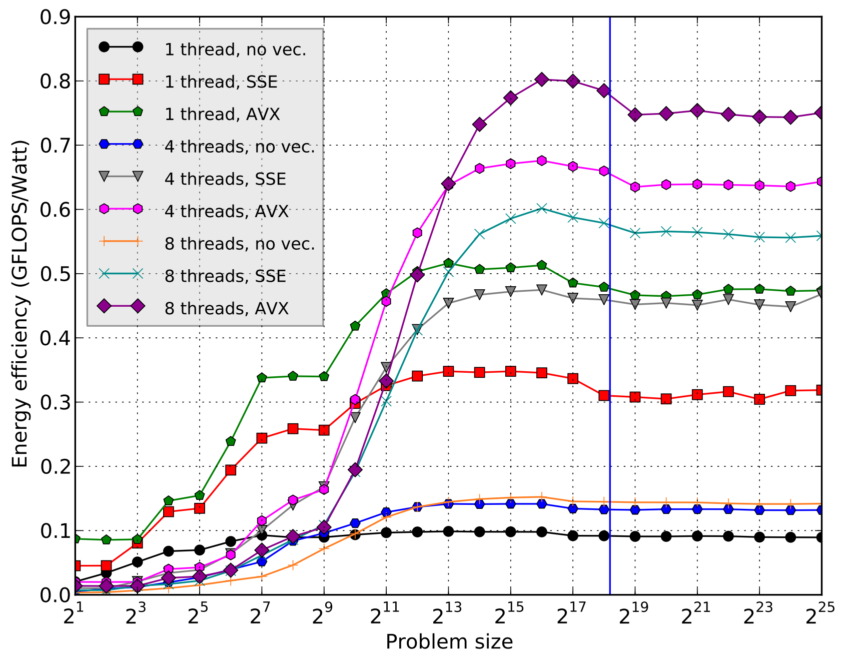


Fig. 3. Energy efficiency vs. problem size for Black-Scholes. The vertical line marks the *8MB point*, *i.e.* the problem size where application data require the entire LLC.

of Frigo and Johnson [13]. For problem sizes up to $N = 2^{14}$, energy efficiency is higher for vectorized single-thread than for parallel execution, and AVX provides further benefits over SSE. It is interesting to note that the intersection coincides with L2 cache size. For the last-level cache problem sizes of 2^{14} through 2^{18}, 4-thread execution provides higher energy efficiency, in proportion to the speedup.

For matrix multiplication, Figures 6 and 7 show that even though eight threads perform significantly better than one, energy efficiency is lower for all problem sizes due to a higher energy consumption rate. As the ALU and L1/L2 caches are shared between hyperthreads on a single core, the performance using eight threads is lower than with four, because tiled, dense matrix-matrix multiplication is computation bound.

4 Related Work

Duran *et al.* [4] evaluate OmpSs implementations of Black-Scholes and Matrix Multiply, but focus on performance only. Comparing with their 4-core result, we get a performance improvement in excess of factor 10. We attribute the difference to the higher CPU clock frequency of our test system, and AVX vectorization.

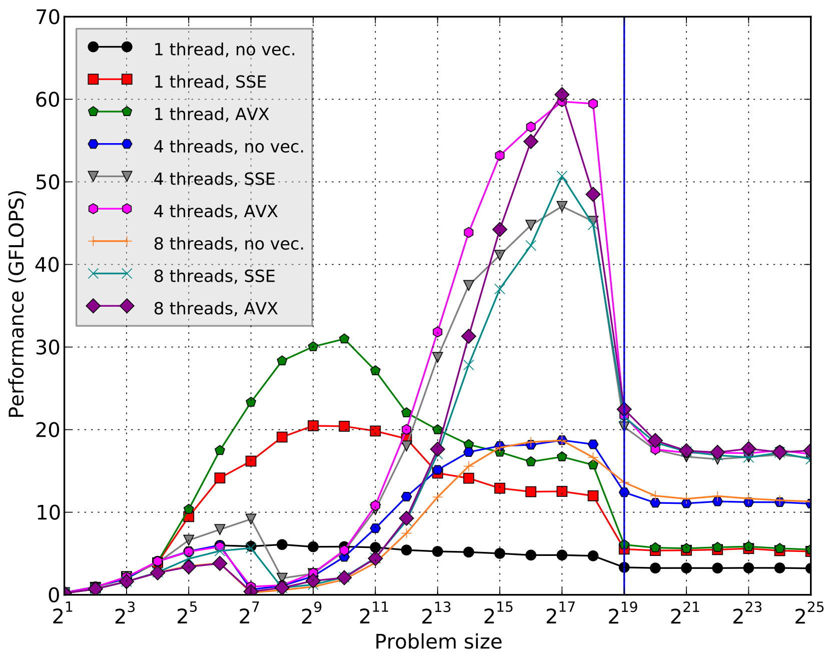


Fig. 4. Performance vs. problem size for FFTW. The *8MB point* is marked by the vertical line.

Ge *et al.* [15] show how the PowerPack framework can be used to study in depth the energy efficiency of parallel applications on clusters with multi-core nodes. The framework is measurement based, and can be used to identify the energy consumption of all major system components.

Li and Martinez [16] develop and use an analytical model of the power-performance implications of degree of parallelism and voltage/frequency scaling. They confirm their analytical results by detailed simulation.

Molka *et al.* [17] discuss weaknesses of the Green500 list with respect to ranking HPC system energy efficiency. They introduce their own benchmark using a parallel workload generator to stress main components in a HPC system.

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

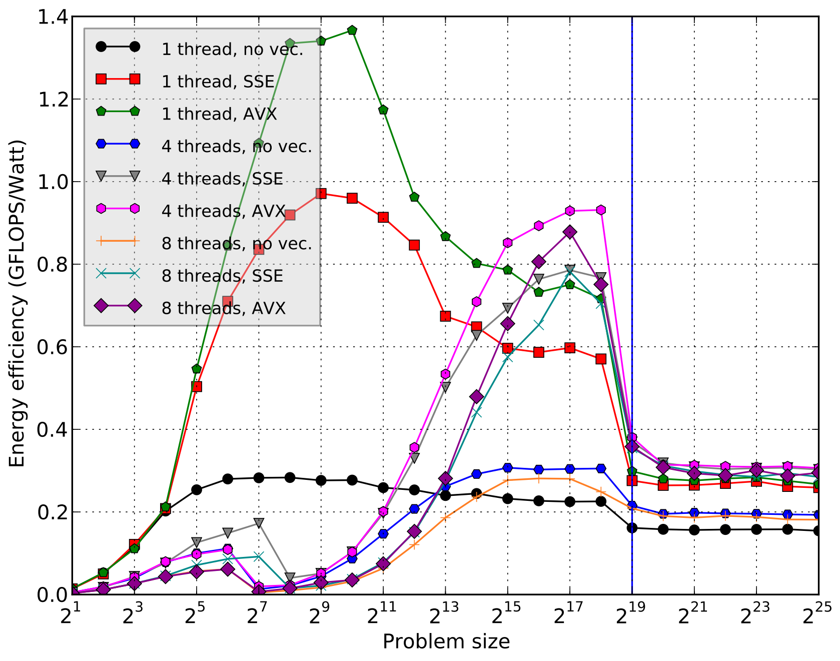


Fig. 5. Energy efficiency vs. problem size for FFTW. The *8MB point* is marked by the vertical line.

5 Conclusions and Future Work

Using chip energy performance counters to instrument three floating-point intensive benchmarks, our experiments show that vectorization provides a significant improvement in on-chip energy efficiency, and that energy efficiency varies with problem size in common application kernels. In our results we have seen that vectorization improves both performance and energy efficiency, while the performance improvement from thread parallelism does not necessarily imply a better energy efficiency.

Variation of energy efficiency with task size suggests that energy-aware task scheduling may adapt task sizes for energy efficient execution, which provides an interesting direction for future research. We also plan to extend the work by studying the impact of varying CPU clock frequencies, OmpSs scheduling policies, and using Turbo Boost Technology. We will apply the Intel Energy Checker SDK and Yokogawa WT210 Power analyzer, to refine energy profiles by including off-chip bandwidth and memory system parameters. The experiments will be extended to a SGI Altix ICE X supercomputer, featuring 2×8 Sandy Bridge multi-core processors.

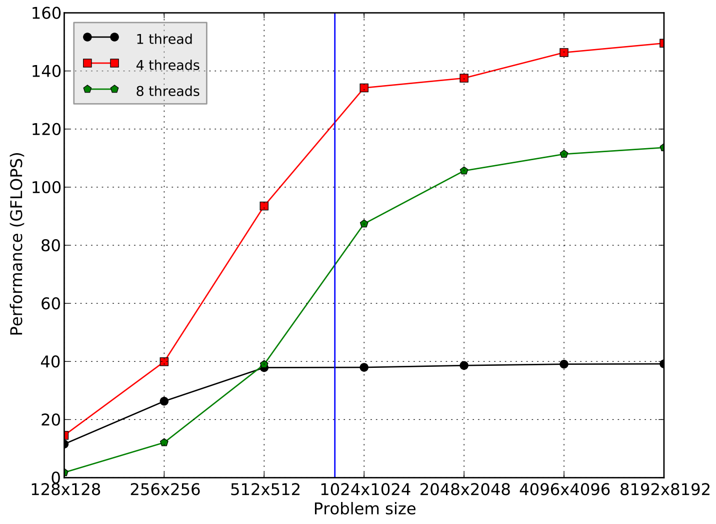


Fig. 6. Performance in MFLOPS of matrix multiplication for different problem sizes. The *8MB point* is marked by the vertical line.

Acknowledgments. The authors gratefully acknowledge the support of the PRACE 2IP project, the NOTUR project, the HiPEAC Network of Excellence, and the help from OmpSs researchers at UPC in Barcelona.

References

1. "Mont Blanc project website." http://www.montblanc-project.eu/.
2. "The Green 500 - Ranking the World's Most Energy Efficient Supercomputers." http://www.green500.org.
3. J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE Int'l Conf. on*, pp. 142 –151, oct 2008.
4. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architetcures," *Parallel Processing Letters*, vol. 21, pp. 173–193, Mar. 2011.
5. A. Ramirez, "European scalable and power efficient HPC platform based on low-power embedded technology." http://www.eesi-project.eu/, Oct 2011. Presentation at the EESI conference.
6. R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 12, pp. 3–35, 2001.
7. Intel, *Intel®64 and IA-32 Architectures Optimization Reference Manual*, jun 2011.
8. Intel, *Avoiding AVX-SSE Transition Penalties*, nov 2011.
9. S. Rivoire, M. Shah, P. Ranganatban, C. Kozyrakis, and J. Meza, "Models and metrics to enable energy-efficiency optimizations," *Computer*, vol. 40, pp. 39 –48, Dec. 2007.

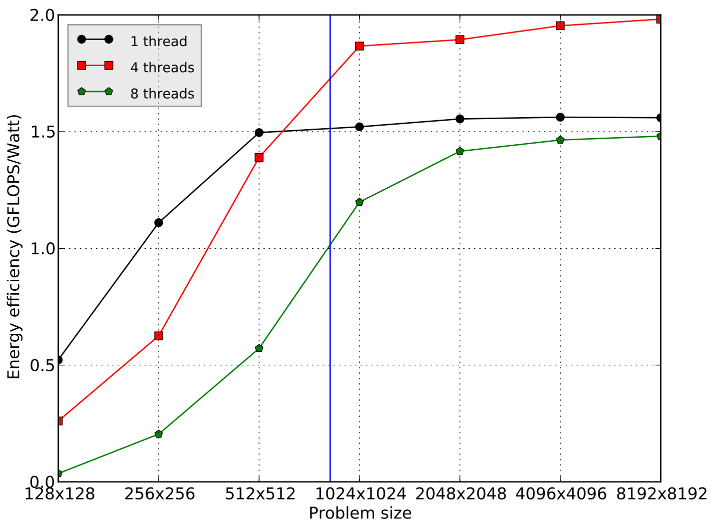


Fig. 7. Energy efficiency in MFLOPS/watt of matrix multiplication for different problem sizes. The *8MB point* is marked by the vertical line.

10. H. Lien, "Case Studies in Multi-core Energy Efficiency of Task Based Programs (preliminary title)," Master's thesis, Norwegian University of Science and Technologoy, Trondheim, Norway, 2012. Work in progress, to be submitted July 2012.
11. Intel, *Intel®64 and IA-32 Architecture Software Development Manual*, Dec 2011.
12. C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proc. of the 17th int'l conf. on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 72–81, 2008.
13. M. Frigo and S. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, pp. 216 –231, feb. 2005.
14. S. L. Moshier, "Cephes Math Library." http://www.netlib.org/cephes.
15. R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, pp. 658 –671, may 2010.
16. J. Li and J. F. Martínez, "Power-performance considerations of parallel computing on chip multiprocessors," *ACM Transactions on Architecture and Code Optimization*, vol. 2, pp. 397–422, Dec. 2005.
17. D. Molka, D. Hackenberg, R. Schöne, T. Minartz, and W. Nagel, "Flexible workload generation for HPC cluster efficiency benchmarking," *Computer Science - Research and Development*, pp. 1–9.
18. H. Anzt, M. Castillo, J. Fernández, V. Heuveline, F. Igual, R. Mayo, and E. Quintana-Ortí, "Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors," *Computer Science - Research and Development*, pp. 1–9.

References

[1] A. Ramirez, "European scalable and power efficient HPC platform based on low-power embedded technology." www.eesi-project.eu/media/BarcelonaConference/Day2/13-Mont-Blanc_Overview.pdf, oct 2011. [cited at p. xv, 6, 7, 10, 12]

[2] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, Hill, K., and Others, "Exascale computing study: Technology challenges in achieving exascale systems," tech. rep., University of Notre Dame, CSE Dept., 2008. [cited at p. 1, 5, 6]

[3] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency," white paper, Intel, March 2008. [cited at p. 2, 12]

[4] "Top500 Supercomputer Sites." http://www.top500.org. [cited at p. 5, 6]

[5] "EESI Website." http://www.eesi-project.eu. [cited at p. 5]

[6] C. Carvalho, "The Gap between Processor and Memory Speeds," 2002. [cited at p. 5]

[7] F. Baude, D. Caromel, N. Furmento, and D. Sagnol, "Optimizing metacomputing with communication-computation overlap," in *Parallel Computing Technologies* (V. Malyshkin, ed.), vol. 2127 of *Lecture Notes in Computer Science*, pp. 190–204, Springer Berlin / Heidelberg, 2001. [cited at p. 5]

[8] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 58–, IEEE Computer Society, 2005. [cited at p. 5]

[9] S. B. Baden and S. J. Fink, "Communication overlap in multi-tier parallel algorithms," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, (Washington, DC, USA), pp. 1–20, IEEE Computer Society, 1998. [cited at p. 5]

[10] P. Cicotti and S. Baden, "Short paper: Asynchronous programming with tarragon," in *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pp. 375 –376, 0-0 2006. [cited at p. 5]

[11] "The Green 500 - Ranking the World's Most Energy Efficient Supercomputers." http://www.green500.org. [cited at p. 6, 15, 18]

[12] W. Saunders, "Rethinking Supercomputer Performance and Efficiency for Exascale." http://communities.intel.com/community/openportit/server/blog/2011/10/20/rethinking-supercomputer-performance-and-efficiency-for-exascale. [cited at p. 6]

[13] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –8, may 2009. [cited at p. 6]

[14] AnandTech, "Samsung's Galaxy S II Preliminary Performance: Mali-400MP Benchmarked." http://www.anandtech.com/show/4177/samsungs-galaxy-s-ii-preliminary-performance-mali400-benchmarked. [cited at p. 7]

[15] "Mont Blanc project website." http://www.montblanc-project.eu/. [cited at p. 7]

[16] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3 – 35, 2001. [cited at p. 7, 23]

[17] M. Frigo and S. G. Johnson, "The Fastest Fourier Transform in the West," in *the Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing, ICASSP '98*, 1997. [cited at p. 7, 18, 46]

[18] HDF Group, "HDF Web site." http://www.hdfgroup.org/. [cited at p. 7]

[19] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: a portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, (Los Alamitos, CA, USA), pp. 2–11, IEEE Computer Society Press, 1990. [cited at p. 7]

[20] W. BOSMA, J. CANNON, and C. PLAYOUST, "The magma algebra system i: The user language," *Journal of Symbolic Computation*, vol. 24, no. 3–4, pp. 235 – 265, 1997. [cited at p. 7]

[21] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili, "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, p. 35, nov. 2005. [cited at p. 7]

[22] ScaleMP, "vsmp architecture." http://www.scalemp.com/architecture. [cited at p. 7]

[23] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010. [cited at p. 8]

[24] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," in *Proceedings of the international conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 214–224, ACM, 2011. [cited at p. 8]

[25] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES," *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011-03-01 2011. [cited at p. 8, 10, 17]

[26] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Productive Cluster Programming with OmpSs," in *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part I* (E. Jeannot, R. Namyst, and J. Roman, eds.), vol. 6852 of *Lecture Notes in Computer Science*, pp. 555–566, Springer, 2011. [cited at p. 10]

[27] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of openmp task scheduling strategies," in *OpenMP in a New Era of Parallelism* (R. Eigenmann and B. de Supinski, eds.), vol. 5004 of *Lecture Notes in Computer Science*, pp. 100–110, Springer Berlin / Heidelberg, 2008. [cited at p. 11, 17]

[28] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. [cited at p. 11]

[29] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, mar 1995. [cited at p. 11]

[30] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, mar 2005. [cited at p. 11, 49]

[31] C. Meenderinck and B. Juurlink, "Euro-par 2008 workshops - parallel processing," in *Euro-Par 2008 Workshops - Parallel Processing* (E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, eds.), ch. (When) Will CMPs Hit the Power Wall?, pp. 184–193, Berlin, Heidelberg: Springer-Verlag, 2009. [cited at p. 11, 49]

[32] Intel, *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*, mar 2004. [cited at p. 11, 49]

[33] Nvidia, *Fermi Compute Architecture Whitepaper*. [cited at p. 12]

[34] ARM, "Mali-T658." http://www.arm.com/products/multimedia/mali-graphics-hardware/mali-t658.php. [cited at p. 12]

[35] Intel, *Extending the World's Most Popular Processor Architecture*, 2006. [cited at p. 12]

[36] Intel, *Avoiding AVX-SSE Transition Penalties*, nov 2011. [cited at p. 13, 14]

[37] ARM, "ARM NEON." http://www.arm.com/products/processors/technologies/neon.php. [cited at p. 14]

[38] ARM, "ARM Advanced SIMD Instruction Scheduling." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0409h/Babfjcjb.html. [cited at p. 14]

[39] S. Rivoire, M. Shah, P. Ranganatban, C. Kozyrakis, and J. Meza, "Models and metrics to enable energy-efficiency optimizations," *Computer*, vol. 40, pp. 39 –48, dec. 2007. [cited at p. 15]

[40] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pp. 8 –11, oct 1994. [cited at p. 15, 16]

[41] A. J. Martin, M. Nyström, and P. I. Pénzes, "Power aware computing," in *Power Aware Computing* (R. Graybill and R. Melhem, eds.), ch. ET2: A Metric For Time and Energy Efficiency of Computation, pp. 293–315, Norwell, MA, USA: Kluwer Academic Publishers, 2002. [cited at p. 16]

[42] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, pp. 1277 –1284, sep 1996. [cited at p. 16]

[43] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The energy efficiency of CMP vs. SMT for multimedia workloads," in *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, (New York, NY, USA), pp. 196–206, ACM, 2004. [cited at p. 17]

[44] E. Grochowski and M. Annavaram, "Energy per Instruction Trends in Intel®Microprocessors," *Technology@Intel Magazine*, 2006. [cited at p. 17]

[45] M. Frigo and S. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, pp. 216 –231, feb. 2005. [cited at p. 18]

[46] M. Frigo and S. G. J. et al, "FFT Benchmark Results." http://www.fftw.org/speed/. [cited at p. 18]

[47] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of iram architectures," in *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pp. 327 –337, jun 1997. [cited at p. 18]

[48] "The Green 500 Run Rules." http://www.green500.org/docs/pubs/RunRules_Ver0.9.pdf. [cited at p. 18]

[49] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *Micro, IEEE*, vol. 31, pp. 86 –95, march-april 2011. [cited at p. 18]

[50] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proc. of the 17th int'l conf. on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 72–81, 2008. [cited at p. 19]

[51] F. Black and M. S. Scholes, "The Pricing of Options and Corporate Liabilities," *Journal of Political Economy*, vol. 81, pp. 637–54, May-June 1973. [cited at p. 19, 20]

[52] L. R. Rabiner, B. Gold, and C. K. Yuen, *Theory and Application of Digital Signal Processing*. Prentice-Hall, feb. 1978. [cited at p. 20]

[53] R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," *J. ACM*, vol. 12, pp. 95–113, January 1965. [cited at p. 20]

[54] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, pp. 281–292, 1971. 10.1007/BF02242355. [cited at p. 20]

[55] E. O. Brigham and R. E. Morrow, "The fast Fourier transform," *Spectrum, IEEE*, vol. 4, pp. 63 –70, dec. 1967. [cited at p. 21]

[56] Intel, *Intel®64 and IA-32 Architectures Software Developer's Manual*, dec 2011. [cited at p. 25, 26, 41, 43, 44]

[57] ARM, "SYS_CFG Registers Documentation." http://infocenter.arm.com/help/topic/com.arm.doc.dui0447e/CACDEFGH.html. [cited at p. 26]

[58] ARM, "CoreTile Express A9x4 Voltage, Current and Power Monitoring." http://infocenter.arm.com/help/topic/com.arm.doc.dui0448e/CHDCEIEJ.html. [cited at p. 27]

[59] ARM, "ARM Legacy Memory Map." http://infocenter.arm.com/help/topic/com.arm.doc.dui0447e/ch04s02s01.html. [cited at p. 27]

[60] ARM, "ARM System Register Summary." http://infocenter.arm.com/help/topic/com.arm.doc.dui0447e/CHDICIIF.html. [cited at p. 27]

[61] S. L. Moshier, "Cephes Math Library." http://www.netlib.org/cephes. [cited at p. 28, 45]

[62] J. Pommier, "Simple SSE and SSE2 (and now NEON) optimized sin, cos, log and exp." http://gruntthepeon.free.fr/ssemath/. [cited at p. 28]

[63] Intel, *Intel®Advanced Vector Extensions Programming Reference*, jun 2011. [cited at p. 29]

[64] Intel, "Haswell New Instruction Descriptions Now Available!." http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/. [cited at p. 29]

[65] ARM, "VRECPE and VRSQRTE Instructions Documentation." http://infocenter.arm.com/help/topic/com.arm.doc.dui0489g/CIHCHECJ.html. [cited at p. 33]

[66] ARM, "VRECPS and VRSQRTS Instructions Documentation." http://infocenter.arm.com/help/topic/com.arm.doc.dui0489g/CIHDIACI.html. [cited at p. 33]

[67] IEEE, “IEEE Standard for Floating-Point Arithmetic,” tech. rep., Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, Aug. 2008. [cited at p. 33]

[68] Matteo Frigo and Stevenj G. Johnson, “benchFFT.” http://www.fftw.org/benchfft/. [cited at p. 35]

[69] Intel, *Intel®64 and IA-32 Architectures Optimization Reference Manual*, jun 2011. [cited at p. 37]

[70] ARM, “CoreTile Express A9x4 Test Chip Clocks.” http://infocenter.arm.com/help/topic/com.arm.doc.dui0448e/CHDEJGAD.html. [cited at p. 39]

[71] ARM, “CoreTile Express A9x4 specification sheet.” http://www.arm.com/files/pdf/CE_A9x4(1).pdf. [cited at p. 39]

[72] ARM, “ARMv8 Technology Preview.” http://www.arm.com/files/downloads/ARMv8_Architecture.pdf. [cited at p. 42]

[73] Intel, “Intel Core i7-2600K Processor specifications.” http://ark.intel.com/products/52214/Intel-Core-i7-2600K-Processor-(8M-Cache-3_40-GHz). [cited at p. 44]

[74] University of Tennessee, “PAPI Website.” http://icl.cs.utk.edu/papi/index.html. [cited at p. 44]

[75] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, “Effective automatic parallelization of stencil computations,” *SIGPLAN Not.*, vol. 42, pp. 235–244, June 2007. [cited at p. 45]

[76] Stanford University, “Folding@home FLOP FAQ.” http://folding.stanford.edu/English/FAQ-flops. [cited at p. 45, 46]

[77] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967. [cited at p. 49]

[78] M. Frigo and S. G. Johnson, *FFTW User’s Manual*, March 2003. [cited at p. 62]