NTNU – Trondheim
Norwegian University of
Science and Technology

Energy Efficiency Studies of Mont Blanc Applications

Mads Holden

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

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

Problem Statement

Energy Efficiency Studies of Mont Blanc Applications

The main goal of this master thesis project is performance and energy-efficiency studies of one or more Mont Blanc application kernels. Mont Blanc is an ongoing EU - 7FP research project aiming at developing prototypes of future exascale supercomputers. The Mont Blanc (MB) project will use the OmpSs programming system developed in Barcelona, and plans to use Mali GPUs as accellerators programmed in OpenCL.

Since it is unclear when will we get access to OpenCL for the Mali GPU, and also when OpenCL is well integrated with OmpSs, the CARD group will seek alternative execution platforms providing OpenCL. Likewise, the student should evaluate some MB kernels on some relevant software that is available, and having the main goal of the master thesis project in mind.

It is of particular interest to the CARD group to study how different cores can be used in parallel solutions to save energy or how other implementation choices will affect both execution time and energy consumption.

Acknowledgements

I would like to thank my supervisor Lasse Natvig for giving me excellent supervision and motivation during the writing of this thesis.

I would also like to thank my two co-supervisors, Asbjørn Djupdal and Juan Cebrian, for both technical and non-technical help.

The figures in this thesis were made with the python library matplotlib [1], and the tables with the python library Jinja2 [2]. The report was written in LaTeX.

Abstract

In this thesis, the performance and energy efficiency of four different implementations of matrix multiplication, written in OmpSs and OpenCL, is tested and evaluated. The benchmarking is done using an Intel Ivy Bridge Core i7 3770K. The results are evaluated and discussed with regards to different optimization configurations, like vectorization and multi-threading. Energy measurements are taken using PAPI, which in turn uses the Running Average Power Limit interface in the Intel processor to take energy readings. Performance is presented using MFLOPS, while energy efficiency is compared using MFLOPS/W, watts used, and the energy delay product and energy delay squared.

The OpenCL versions are compared with and without vectorization. One of the applications using OmpSs is also measured with regards to vectorization, and also number of threads. The last OmpSs version uses the BLAS implementation ATLAS, which is already vectorized. Therefore it is only compared using number of threads.

SSE and AVX vectorization is shown to significantly improve performance while using little to no extra energy per second for all implementations. Multi-threading also gives higher performance, however this consumes more energy. Running with eight threads was shown to spend more energy while performing worse when using ATLAS. The OmpSs version using ATLAS was both the fastest and most energy efficient, peaking at 125 GFLOPS and 2.7 GLOPS/W while running with four threads and using AVX.

Abstract (Norwegian)

I denne oppgaven blir ytelsen og energi-effektiviteten av fire forskjellige implementasjoner av matrisemultiplikasjon, skrevet i OmpSs og OpenCL, testet og evaluert. Målingene blir gjort på en Intel Ivy Bridge Core i7 3770K. Resultatene blir evaluert og diskutert med tanke på forskjellige optimaliseringskonfigurasjoner, som vektorisering og multitråding. Energimålingene ble tatt ved hjelp av PAPI, som igjen bruker Running Average Power Limit grensesnittet i Intel prosessoren for å lese energibruken. Ytelsen blir presentert i MFLOPS, og energieffektiviteten blir sammenlignet med MFLOPS/W, watt, og energy delay product og energy delay squared.

OpenCL-versjonene blir sammenlignet med og uten vektorisering. En av programmene som bruker OmpSs blir også målt med tanke på vektorisering, i tillegg til antall tråder. Den siste OmpSs-versjonen bruker BLAS-implementasjonen ATLAS, som allerede er vektorisert. Derfor blir den kun sammenlignet med tanke på antall tråder.

SSE- og AVX-vektorisering viste seg å øke ytelsen betydelig samtidig som det bruker lite til ingen ekstra energi per sekund for alle implementasjoner. Multitråding gir også høyere ytelse, men bruker i tillegg mer energi. Å kjøre med åtte tråder viste seg å bruker mer energi å yte værre når ATLAS ble brukt. OmpSs-versjonen som brukte ATLAS var både raskest og mest energieffektiv, og kom opp i 125 GFLOPS og 2.7 GFLOPS/W, kjørende med fire tråder og AVX.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will present the scope and outline of the thesis. Some terminology will also be explained.

1.1 Thesis Scope

The goal of this thesis is to evaluate the performance and energy usage of different implementations of matrix multiplication. The applications are developed using OpenCL and OmpSs, which will be compared both theoretically and in the results of the tests. The different applications will be tested and measured using different optimizations, including vectorization and multithreading. Because there were at the time of writing no OpenCL drivers available for the Mali GPU, another computer capable of running OpenCL was chosen. The computer used in benchmarking is the Intel Ivy Bridge Core i7 3770k. Again because there are no OpenCL drivers released for the on-board GPU of the Ivy Bridge, only the CPU will be tested.

1.2 Terminology

Hyper-threading is Intel's proprietary simultaneous multithreading implementation. For each processor core that is physically present, the operating

system addresses two logical cores, that share the workload whenever possible. In this thesis hyper-threaded applications will refer to applications that run using eight threads, as there are four physical cores on the target hardware.

FLOPS written with capital letters will in this thesis refer to floating point operations per second, while a flop (and its plural flops), written in lower case, will refer to a floating point operation, e.g. one addition or multiplication. Simply put, $FLOPS = flops/s$.

1.3 Thesis Outline

The thesis is divided into six chapters and three appendices. This is chapter one, which introduces the work. Chapter two covers the background of topics important to the thesis, presenting the Mont Blanc project, OpenCL, OmpSs, matrix multiplication, energy measurement and metrics, and related work. Chapter three presents the implementation of the different applications tested, and the energy measurement. Chapter four shows the exeriment setup and methodology, going through both the hardware and software used to benchmark the applications. Chapter five covers the results of the tests run, while discussing and comparing performance and energy of the different applications. Chapter six concludes the report.

Chapter 2

Background

2.1 Exascale computing and the Mont Blanc Project

A major milestone for supercomputers is to reach one exaFLOPS. The most powerful supercomputer as of November 2012 is the Titan [3], from the Oak Ridge National Laboratory in the United States. It has been measured at 17.59 petaFLOPS, which means a big improvement is necessary to reach the milestone. The Mont Blanc project is an EU project located at the Barcelona Supercomputing Center, which intends to address some of the challenges [4].

2.1.1 Exascale computing

Supercomputers have shown an exponential performance increase over time. A tenfold improvement in performance is observed every 3.6 years, which means that exascale performance should be attainable by 2018 [5]. However, the power requirements, assuming they increase by the same factor, would be over 400 MW. A more realistic power budget would be 20 MW [6, p. 8], which would require an energy efficiency of 50 GFLOPS/W. The Green500 list [7] ranks supercomputers based on their energy efficiency.

As of November 2012, the top ranking supercomputer is just shy of 2.5 GFLOPS/W, so a substantial increase in efficiency is required.

Another big issue is the increased complexity of writing software for the increasingly complex supercomputer systems.

2.1.2 The Mont Blanc Project

The Mont Blanc project takes a new approach to the issues arising from power efficiency in exascale computing. The project defines 3 main objectives [4]:

- To deploy a prototype HPC system based on currently available energy-efficient embedded technology, scalable to 50 petaFLOPS using 7 MW. This system should be competitive with Green 500 leaders in 2014.
- To design a next-generation HPC system and new embedded technologies targeting HPC systems that would overcome most of the limitations encountered in the prototype system, scalable to 200 petaFLOPS using 10 MW. This system should be competitive with Top 500 leaders in 2017.
- To port and optimise a small number of representative exascale applications capable of exploiting this new generation of HPC systems.

The prototype system listed in these goals would achieve 7 GFLOPS/W, nearly tripling the effiency of the current leader of the Green 500 list. The second goal is even more ambitious, hoping the reach 20 GFLOPS/W. They plan to achieve these goals by utilizing heterogeneous computing, using a combination of CPUs and GPUs. Embedded power-efficient technology, usually used in mobile devices like cell phones and tablets, will be used in the new systems to reduce costs and increase efficiency. Mobile processors are today 100 times cheaper than a typical server processor, but only 9 times slower [5]. The main compute platform chosen to be used in the Mont Blanc supercomputers is the Samsung Exynos 5 Dual chip [8]. The

Exynos 5 Dual contains a 1.7 GHz dual-core ARM Cortex-A15, and a 533 MHz quad-core ARM Mali-T604 GPU [9]. It has been shown that systems consisting of a combination of CPUs and GPUs are more power-efficient than pure CPU systems [10].

2.2 OpenCL

OpenCL (Open Computing Language) [11] is a widely used framework for writing programs across heterogeneous platforms consisting of CPUs, GPUs, and other processing units. It includes a language for writing kernels, and an API for the C programming language used to define and control the platforms and devices. Initially developed at Apple Computer, it is now maintained as an open standard by the Khronos Group, a non-profit technology consortium. The OpenCL project enjoys support from much of the hardware industry, with drivers and SDKs available from AMD, Apple, ARM, Intel, and others [12].

2.2.1 Usage

An application leveraging OpenCL is written in C or C++ using a series of API calls, used for setting up the environment, work units, and execution queues. The code to be executed on the device is written in a separate file and compiled at runtime. The kernel files are written in a programming language based on a subset of C99 with extensions for parallelism. The kernel langauge omits the use of function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files. The extensions it provides are easy-to-use vector types and operations, synchronization, and many built-in functions [11].

2.2.2 Vectorization

OpenCL was created to make vecorization very easy for the developers. The language extensions mentioned above easily creates SIMD instructions without writing intrinsics or assembly. For instance, let us look at the

SAXPY (Single-precision real Alpha X Plus Y) operation. This operation is a combination of scalar multiplication and vector addition, and is shown in equation 2.1, where α is a scalar, and $\vec{x}$ and $\vec{y}$ are vectors.

$$\vec{y} = \alpha\vec{x} + \vec{y} \tag{2.1}$$

A simple version might look like the code in listing 2.1. This would of course be quite slow, only operating on a single vector index at a time.

Listing 2.1 Simple SAXPY implementation

```
1 void saxpy_simple(float a, float *x, float *y, int n) {
2     for (int i = 0; i < n; i++) {
3         y[i] += a*x[i];
4     }
5 }
```

Instead we could use intrinsics, like in listing 2.2, which uses Intel AVX intrinsics. This is in theory eight times faster than the simple implementation, but is difficult to read and write, and it only works on Intel hardware.

Listing 2.2 SAXPY implementation using AVX intrinsics

```
1 void saxpy_intrinsics(float a, float *x, float *y, int n) {
2     __m256 _a = _mm256_set1_ps(a);
3     for (int i = 0; i < n; i+=8) {
4         __m256 _x = _mm256_loadu_ps(&x[i]);
5         __m256 _y = _mm256_loadu_ps(&y[i]);
6         _y = _mm256_add_ps(_y, _mm256_mul_ps(_a, _x));
7         _mm256_storeu_ps(&y[i], _y);
8     }
9 }
```

The OpenCL version shown in listing 2.3 is strikingly similar to the simple version. However, because it operates on the `float8` vector type, it is theoretically as fast as the code using intrinsics. The OpenCL compiler will generate the proper SIMD operations depending on the hardware. Note

that only the OpenCL kernel code is shown, not the host code, which is significally longer and more verbose. Another option yet is to write an OpenCL kernel using regular `float`s, and let the compiler automatically vectorize the code.

Listing 2.3 SAXPY implementation using OpenCL

```
1 __kernel void saxpy_opencl(float a, __global float8 *x, __global float8 *y, int n)
      {
2     for (int i = 0; i < n/8; i++) {
3         y[i] += a * x[i];
4     }
5 }
```

2.3 OmpSs

OmpSs (Open Multi-Processing Super Scalar) is a set of extensions to OpenMP. OpenMP is a popular framework for multithreaded applications in shared memory computers. Compared to other parallel processing frameworks like pthreads or MPI, OpenMP is fairly easy to use, requiring far less code to add multithreading to an application. It is used for the most part by using preprocessor directives in the form of pragmas. There is also a C API with functions used for setting up the number of threads, getting thread ids, et cetera. Despite its ease of use, it has some limitations. It can not be used for GPUs or other accelerators, and it can not be used across a cluster of computers.

GPUs are today usually programmed using OpenCL or CUDA. These programming frameworks are quite complicated to use, requiring the developer to manually manage data movement, memory management, and error handling. CUDA is also proprietary, and only works for GPUs from Nvidia [13]. For cluster programming, MPI has been the *de facto* standard for many years, however it also requires a lot of manual work for the developer.

OmpSs is a programming model based on OpenMP which is developed at the Barcelona Supercomputing Center [14]. It aims to provide a parallel programming framework that will be able to exploit heterogeneous computing systems and computing clusters. Being an extension to OpenMP, it keeps the simplicity of its predecessor, keeping the high developer productivity, while providing powerful innovations.

The execution model differs slightly from OpenMP. While OpenMP has a fork-join model, OmpSs uses a thread-pool model where all the threads exist from the beginning of the execution. Only one of those threads, the master thread, executes user code while the other threads remain ready to execute work when available [15]. As a team of threads exists from the beginning, there is no need for an explicit `parallel` directive, which is deprecated in OmpSs.

OmpSs consists of two parts. Mercurium is a C/C++ source-to-source compiler, transforming high-level directives into a parallelized version of the application. Nanos++ is a runtime library providing the parallel services to manage task creation, synchronization, and data movement [16].

2.3.1 Task-based programming

OpenMP was initially focused on loop parallelism, but was extended with task based parallelism in version 3.0 [17]. Loop-based parallelism means that the loop iterations are simply split between the threads. In task-based parallelism the programmer can specify tasks, and later ensure that all tasks defined up to some point have finished. When called, each task is assigned to a thread from a thread pool by a task scheduler.

The main advantage of using tasks is that the parallelization becomes more dynamic. Work in a merge-sort implementation is for example generated recursively. This makes it very hard to parallelize using loops, but is quite easily expressed by defining each recursive call as a task, as shown in listing 2.4 (merge code omitted for brevity).

OmpSs expands the task abstraction found in OpenMP by introducting data-based dependencies between tasks [15]. Three new clauses are added to the task directive: `input`, `output` nad `inout`. All three accept an expression

Listing 2.4 Merge sort using OmpSs

```
1  #pragma omp task inout(array[low:high])
2  void merge(int *array, int low, int mid, int high);
3
4  #pragma omp task inout(array[low:high])
5  void _merge_sort(int *array, int low, int high) {
6      if (low < high) {
7          int mid = (low + high) / 2;
8
9          _merge_sort(array, low, mid);
10         _merge_sort(array, mid+1, high);
11
12         merge(array, low, mid, high);
13     }
14 }
15
16 void merge_sort(int *array, int length) {
17     _merge_sort(array, 0, length);
18     #pragma omp taskwait
19 }
```

that must evaluate to a set of *lvalues*. Tasks with an `input` clause will not be eligible to run as long as a previously created task with an `output` clause with the same *lvalue* has not finished its execution. The `inout` clauses count as both an `input` and an `output` clause.

2.3.2 Heterogeneous extensions

By default OmpSs generates code for traditional symmetric multiprocessors, like OpenMP does. However, a new directive is introduced in OmpSs which enables execution on heterogeneous systems. The `target` directive specifies that a given element can be run on a specific set of devices. The currently supported devices are SMP, CUDA, Cell, and OpenCL. The developer has to write the device-specific code, but data movement is reduced to simple clauses in the `target` directive, as shown in listing 2.5.

Additionally, OmpSs has a clause to ease splitting up the problem between devices. Listing 2.6 (may need to be split into different files) shows

Listing 2.5 OpenCL kernel called with OmpSs

```
#pragma omp target device(opencl) ndrange(1, global_size, local_size) copy_deps
#pragma omp task in(a[block_size], b[block_size]) out(c[block_size])
__kernel void add(float *a, float *b, float *c, int block_size) {
    for (int i = 0; i < block_size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

the `implements` clause, which specifies that the code is an alternate implementation of a function. This means that the runtime environment will handle the complexities of using multiple devices, while the developer just has to write the device-specific code.

Listing 2.6 Different implementations of functions in OmpSs

```
#pragma omp task in(a[block_size], b[block_size]) out(c[block_size])
void add_block(float *a, float *b, float *c, int block_size)
{
    // plain C kernel code
}
#pragma omp target device(opencl) copy_deps implements(add_block)
__kernel void add_block_cl(float *a, float *b, float *c, int block_size)
{
    // OpenCL kernel code
}
#pragma omp target device(cuda) copy_deps implements(add_block)
__global__ void add_block_cuda(float *a, float *b, float *c, int block_size)
{
    // CUDA kernel code
}
```

2.3.3 OmpSs - OpenCL interoperability

As explained in the previous section, OmpSs supports the OpenCL target. However, using this directive is quite a bit harder than seen in listing 2.5.

The master's thesis of Guillén Allés, Moisés [18] (in spanish) explains how to install and use a version of Mercurium and Nanos++ that is able to compile and run OmpSs applications using OpenCL kernels. Following the advice of my supervisor, I tried translating and following the instructions. After following them the OmpSs applications compiled fine (using the compiler flag `--opencl-code-file=FILE`), but running them only produced segmentation faults.

Near the end of the timeframe of this thesis, a precompiled version of Mercurium and Nanos++ were provided by BSC, along with example code using OpenCL kernels. However, these examples failed to compile using the provided compiler (with error message `parameter '--opencl-code-file =matmul_kernel.cl' ignored`).

2.3.4 Task scheduling

Using OmpSs, all threads are started when the application starts. The distribution of tasks to threads, including the order of execution, is up to the task schedule policy. One of the threads, called the main thread, starts running the application serially. When it encounters a task directive, the task is put into a shared task pool, which is used by the scheduler to divide work between the rest of the threads.

The default scheduler in Nanos++ uses a depth-first algorithm, however many schedulers are available through a run-time flag [19].

2.4 Dense Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra, used widely in scientific and other applications. It is also very popular in benchmarking, because of its high floating point operation to memory access ratio. This thesis considers the problem of computing the product

$$C = AB \tag{2.2}$$

of two large, dense, $N \times N$ matrices, without losing generality. The naive way to compute the product, represented by the code in listing 2.7, is simply

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \tag{2.3}$$

Listing 2.7 Naive Matrix Multiplication

```
1 for(i = 0; i < N; i++) {
2     for(j = 0; j < N; j++) {
3         C[i][j] = 0;
4         for(k = 0; k < N; k++) {
5             C[i][j] += A[i][k] * B[k][j];
6         }
7     }
8 }
```

While simple, this code suffers from poor memory locality, resulting in low reuse of data for large matrices. While each iteration in j reuses row i of A, that row may have been evicted from the cache by the time the inner-most loop completes. In addition, the elements of matrix B are accessed columnwise, while the data is stored in row-major order. To remedy the former, one can divide the matrices into smaller blocks that fit into the processor cache, thus utilizing the data fully once it is fetched from main memory. The latter issue can be circumvented by storing matrix B transposed to offer a more cache-friendly representation.

2.5 Energy Measurement

2.5.1 Running Average Power Limit

Running Average Power Limit, or RAPL, is a set of interfaces developed by Intel designed to provide mechanisms to enforce power consumption limits [20]. The RAPL interfaces are available on the Sandy Bridge or

newer microarchitectures, and consist of several non-architectural MSRs, or model-specific registers. An MSR is a control register present in the x86 instruction set. The MSRs in a linux system are, when the msr kernel module is loaded, represented by pseudo-files located in `/dev/cpu/x/msr`, where x is unique for each processor core. Reading and writing is handled by the rdmsr and wrmsr instructions, provided by the msrtools package [21].

The RAPL interfaces are also very well suited for measuring power consumption of short code paths [22]. The registers describe the units and granularity of the measurements, while also containing the actual values. The granularity is defined as 2^{-ESU}, where ESU is the Energy Status Unit part of the `MSR_RAPL_POWER_UNIT` register. The default value is $0b10000 = 16$, giving a granularity of $15.3\mu J$ [20]. The actual value is stored in the register as an unsigned 32-bit integer. This means that when the counter reaches 2^{32} it will wrap around to zero, giving incorrect energy readings.

The Intel Architectures Software Developer's Manual [20] gives an estimated wrap-around time of the energy status register of 60 seconds when the load is high. Examining it closer shows us that it might be far greater than 60 seconds. This is also observed during testing later on. Using the default granularity of $15.3\mu J$, the values represented may be between $0J$ and $\frac{1}{2^{16}} * (2^{32} - 1) \approx 65536J$. Even running at the maximum thermal design power of 77W, the wrap-around time should be $\frac{65536J}{77W} \approx 851s$.

2.5.2 Performance Application Programming Interface

The Performance Application Programming Interface project, or PAPI, specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors [23]. It is a high-level library which reads counters associated with certain events. A set of predefined events is provided, e.g. `PAPI_TOT_INS` measures the total number of completed instructions.

As of version 5.0, PAPI supports measuring and reporting energy values [24]. On Intel platforms, it does this by reading the RAPL registers described in section 2.5.1. The energy readings available on Intel include the energy usage for the total processor package (`PACKAGE_ENERGY`), the en-

ergy usage for all cores including their caches (referred to as power-plane 0 (PP0_ENERGY)), and the usage by the on-board GPU (referred to as power-plane 1 (PP1_ENERGY)) [20]. The on-board GPU will not be used in this thesis. The difference between PP0_ENERGY and PACKAGE_ENERGY is any on-chip controllers, like for instance memory controllers. The whole processor package is of interest to this thesis, therefore we will be using the PACKAGE_ENERGY event.

PAPI will also be used for measuring cache events. The events measured in this thesis are the level 2 accesses, level 2 misses, level 3 accesses, and level 3 misses. The counters measuring level 2 misses and level 3 accesses will always be equal, as a miss in the level 2 cache will always result in an access in the next level. Similarly, one can measure the level 1 misses by looking at the level 2 accesses.

2.6 Energy Efficiency Metrics

Several metrics exist for measuring energy effieciency. The Green 500 list argues for, and uses, GFLOPS per watt as the metric in their comparison [25]. In this thesis, other metrics will also be looked at and used.

2.6.1 GFLOPS/W

FLOPS, or floating point operations per second, is a widely used measure of computer performance. It is especially often used in scientific calculations that heavily use floating point operations. It is for example used by the Top 500 list.

FLOPS/W is used for measuring the energy efficiency of a system, measuring the rate of computation delivered by a computer for every watt of power consumed. Interestingly, the metric FLOPS/W is equivalent to flops/J:

$$\frac{FLOPS}{W} = \frac{flops/s}{J/s} = \frac{flops}{J} \tag{2.4}$$

This means that when the problem size, and therefore also the number of flops, is fixed, FLOPS/W is simply a measurement of the total energy

spent scaled by a constant.

2.6.2 Energy-delay products

Horowitz, Indermaur, and Gonzalez [26] compare different energy efficiency metrics, and propose the energy-delay-product, usually written as EDP. The two obvious choices for low-power metrics, power and energy, are shown to have serious flaws. Power is easy to reduce by reducing the operating frequency. The energy an operation requires can also easily be made smaller by reducing the supply voltage. Both of these reductions will dramatically increase the delay of the operation. The article then proposes the energy-delay-product, defined as $EDP = Energy \times Time$, which is used in this thesis.

A later paper by Martin, Nyström, and Pénzes [27] introduces the energy-delay-squared metric, abbreviated to EDD. They show that when volage scaling is used, the traditional EDP metric is insufficient to compare implementations, while the EDD metric is. Energy delay squared is defined as $EDD = Energy \times Time^2$. This metric is also used and compared in this thesis.

2.7 Related Work

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

In his master's thesis, Lien evaluates the performance and energy efficiency of two hardware platforms, the Intel Sandy Bridge Core i7 and ARM Cortex-A9 MPCore test chip [28]. The thesis covers techniques like vectorization and multi-threading, using three task-based programs for its evaluation of the platforms. The kernels, Black-Scholes, FFTW, and matrix multiplication, are written using OmpSs, and are compared with different configurations on both platforms.

Chapter 3

Implementation

This chapter will present the code measuring performance counters, along with the different implementations of the matrix multiplication kernel.

3.1 Performance measurement

As explained in section 2.5.2, PAPI was used for measuring performance and energy metrics during the benchmarking. To ease the use of the low-level API described in [29], three functions were implemented; `void start_listening (void)`, `void stop_listening(void)`, and `void print_counters(void)`, all using a static struct defined in the same file. The struct holds all the measurement information, and the information needed by PAPI, and is shown in listing 3.2. The initialization code in shown in listing 3.1. Error checking and cache counters are omitted for brevity.

While implementing the energy measurement code, the wrap-around time of the RAPL counters was tested. As discussed in section 2.5.1, Intel Architectures Software Developer's Manual gives an estimated wrap-around time of 60 seconds, while calculating it here gave a wrap-around time of approximately 851 seconds. Testing showed that neither were correct, with seemingly random wrap-around times for different runs. Testing was done at approximately 40 watts, which should give 1638 seconds. However, the

Listing 3.1 PAPI initialization

```
1  void start_listening() {
2      counters.event_set = PAPI_NULL;
3      int event_code = PAPI_NATIVE_MASK;
4      const PAPI_component_info_t *comp_info = NULL;
5      PAPI_event_info_t event_info;
6
7      PAPI_library_init(PAPI_VER_CURRENT);
8      int num_comps = PAPI_num_components();
9
10     int rapl_comp_id;
11     for (rapl_comp_id = 0; rapl_comp_id < num_comps; rapl_comp_id++) {
12         comp_info = PAPI_get_component_info(rapl_comp_id);
13         if (strcmp(comp_info->name, "rapl") == 0) {
14             break;
15         }
16     }
17     PAPI_create_eventset(&counters.event_set);
18
19     PAPI_enum_cmp_event(&event_code, PAPI_ENUM_FIRST, rapl_comp_id);
20     for (counters.num_events = 0; ret == PAPI_OK; counters.num_events++) {
21         PAPI_event_code_to_name(event_code, counters.event_names[counters.
       num_events]);
22         PAPI_get_event_info(event_code, &event_info);
23
24         strncpy(counters.units[counters.num_events], event_info.units,
       PAPI_MIN_STR_LEN);
25         PAPI_add_event(counters.event_set, event_code);
26
27         PAPI_enum_cmp_event(&event_code, PAPI_ENUM_EVENTS, rapl_comp_id);
28     }
29
30     counters.values = calloc(counters.num_events, sizeof(*counters.values));
31
32     counters.start_time = PAPI_get_real_nsec();
33     PAPI_start(counters.event_set);
34 }
```

Listing 3.2 Performance information

```
1  struct counters
2  {
3      int event_set;
4      long long *values;
5      long long start_time;
6      double elapsed_time;
7      char event_names[MAX_RAPL_EVENTS][PAPI_MAX_STR_LEN];
8      char units[MAX_RAPL_EVENTS][PAPI_MIN_STR_LEN];
9      int num_events;
10 };
11 static struct counters counters;
```

values wrapped anywhere between 40 and 400 seconds. One test tried to read the counters every 5 seconds, which should be well inside both Intel's claims and the calculations performed in this thesis. However, this did not produce any difference in results; the counters still wrapped around seemingly randomly.

3.2 OpenCL Matrix Multiplication

Two versions of the matrix multiplication kernel were developed using OpenCL.

The first is quite simple, using one work-item for each element in the resulting matrix. This work item will compute a single element, i.e. the dot product of one whole row in matrix A with one whole column in matrix B.

The other implementation transposes the second matrix before computing the result. This allows for better cache reuse, and better vectorization of the code, as discussed in section 2.4. In general, any transposition of the matrices that will increase the performance of the multiplication should be worth it. Any reordering or transposition of the matrices will have an asymptotic cost of $O(n^2)$, whereas the multiplication has a cost of $O(n^3)$. Like the simple implementation, it delegates one element of the resulting

matrix to one work-item.

Four versions of each implementation is run:

1. No vectorization
2. Using OpenCL automatic vectorization
3. Using explicit float4 vectorization
4. Using explicit float8 vectorization

The `float4` and `float8` vectorizations use OpenCL functions which, on the Intel CPU, map directly to respectively SSE and AVX instructions, as explained in section 2.2.2. It would be quite interesting to know how the compiler optimizes the code when using automatic vectorization. However, I could not find any way to view the generated assembly. There is a hint at a Linux version of the Intel Offline Compiler Standalone Tool in the online Intel OpenCL user guide [30]. However, the website seems to be outdated, as I could not find any other references to it. Following the link given in the user guide only shows information regarding the Windows version.

One interesting thing to note is the use of the dot product. OpenCL includes a built-in function for efficiently calculating the dot product of two vector types [11, p. 264]. However, this is only supplied for the `float` types `float`, `float2`, `float3`, and `float4`. This means that the main operation of the matrix multiplication will not be fully utilizing AVX. A suboptimal dot product implementation for `float8` is shown in listing 3.3. All kernels are available in appendix B.

Listing 3.3 OpenCL implementation of AVX dot product

```
1 float avx_dot(float8 a, float8 b) {
2     return dot(a.hi, b.hi) + dot(a.lo, b.lo);
3 }
```

3.3 OmpSs Matrix Multiplication

Two different OmpSs implementations were tested. One was developed by me during the writing of this thesis. The other was provided by BSC, edited slightly to be usable with the test framework used for the OpenCL versions. Both kernels are available in appendix C.

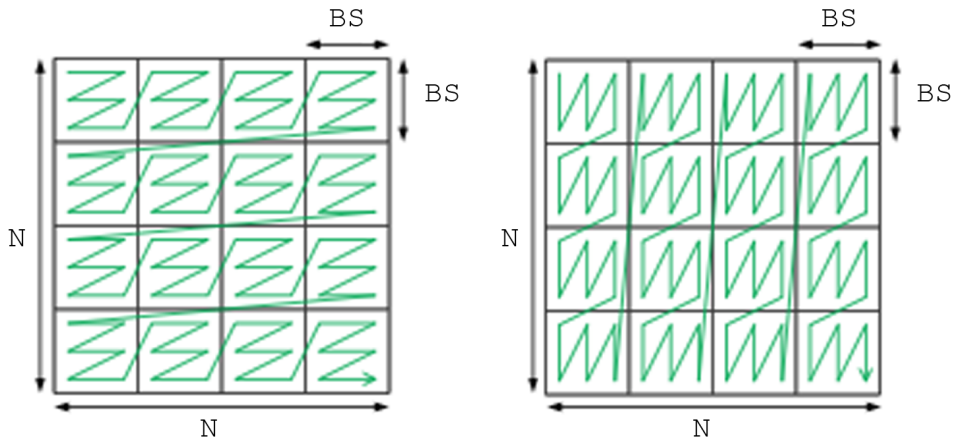
3.3.1 OmpSs

The first implementation uses a technique shown by Matsumoto, Nakasato, and Sedukhin [31]. They arrange the matrices into sub-blocks, using the row-block-row-major layout shown in figure 3.1a. The data for each sub-block of $BS \times BS$ is aligned in row-major order, and the sub-blocks are also aligned in row-major order. This creates $NB = N/BS$ blocks. The implementation used in this thesis uses this arrangement for matrix A (using the notation from equation 2.3). Matrix B is arranged in a column-block-column-major layout, as shown in figure 3.1b. This ensures that all memory lookups are from contiguous memory. The matrices are arranged into blocks to get better cache usage, trying to reuse as much as possible from the two matrices.

The resulting matrix C is divided into blocks, where one OmpSs task calculates one block. Each task then has NB smaller matrix multiplications to calculate, adding the results on the way. This makes a total of $NB * NB$ tasks. One could divide the tasks further, defining each of the smaller matrix multiplications as one task. However, this was shown during experiments do give slightly worse performance.

The application can be compiled to use AVX, SSE, or no vectorization. The code is altered using preprocessor directives, as only small changes were necessary. The vecorization is implemented using Intel intrinsics. When using AVX or SSE, the memory used had to allocated with the POSIX function `posix_memalign` to ensure that the memory is properly aligned. The fastest AVX instructions require memory to be 32-byte aligned, while SSE instructions require the memory to be 16-byte aligned.

The optimal block sizes were first calculated with regards to cache sizes.



(a) Row-block-row-major layout (b) Column-block-column-major layout

Figure 3.1: Matrix data layouts

The application was tested with these block sizes for all optimizations, then adjusted accordingly, and tested again. The block sizes which performed the best for each parameter option were selected.

3.3.2 OmpSs with ATLAS

The BSC version uses the Automatically Tuned Linear Algebra Software (ATLAS) [32] for computing the result. ATLAS is a research project aimed at developing a portably efficient BLAS implementation. It automatically selects the best performing kernels for each BLAS routine. It selects the kernel parameters, like block sizes, using the physical properties of the machine. This is done at compile-time. Because of this, ATLAS has high performance on many different architectures. ATLAS is AVX vectorized on hardware that supports it.

ATLAS is distributed as source code from the webpage [33]. It is also distributed as a package in Ubuntu, installable with the built-in package

manager. For this thesis both were tried, with the source installation being the better choice by far.

This implementation also divides the matrices into blocks for each task to calculate, changing the layout to row-column-row-major, shown in figure 3.1a. One task is here defined as any two blocks in matrices A and B, creating $NB * NB * BS = NB * N$ tasks. Any two blocks that do not share a block in C can be multiplied in parallel. The tasks are then fed into the `cblas_sgemm` function to be calculated by the ATLAS library. ATLAS may in turn sub-divide each block to make them fit into the CPU caches.

Chapter 4

Experiment Setup and Methodology

4.1 Test Bench

The experiments were run on `minvilje`, a desktop computer with an Intel Ivy Bridge Core i7 quad-core processor.

4.1.1 Hardware

`minvilje` has an Intel Core i7-3770K CPU with four cores, and a clock speed of 3.50GHz. The general hardware specifications for `minvilje` can be seen in table 4.1, and the cache specifications are listed in table 4.2. CPU information was collected from `/proc/cpuinfo`, memory information from the tool `dmidecode`, and cache information from `/sys/devices/system/cpu/cpu0/cache/indexX`. A diagram of the Ivy Bridge CPU with caches is shown in figure 4.1

4.1.2 Software

For `minvilje` Ubuntu 12.04.2 LTS was used, running Linux kernel 3.6.0. The third-party software and libraries used for the benchmarking is listed in

Property	Value
CPU model	Intel Core i7-3770K
Model	58
Stepping	9
Clock frequency	3.50GHz
Number of physical cores	4
Number of logical cores	8
Main memory size	16GB
Type	DDR3
Clock speed	1333MHz
Arrangement	4×4GB

Table 4.1: Hardware specifications for minvilje

Cache level	Size	Line size
Level 1	32K (Data) 32K (Instruction)	64B
Level 2	256K	64B
Level 3	8MB	64B

Table 4.2: Cache specifications for minvilje

table 4.3.

4.1.3 Compilation

The OpenCL applications were compiled using GCC, while the OmpSs applications used Mercurium (mcc). The compiler flags are given in table 4.4.

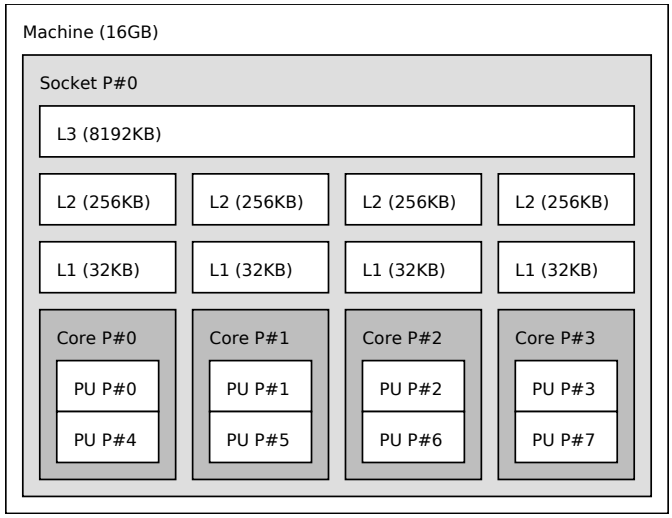


Figure 4.1: Ivy Bridge CPU with caches

Developer	**Name**	**Version**
The GNU Project	gcc	4.6.3
ICL, University of Tennessee	PAPI	5.1.0.2
Intel Corporation	Intel OpenCL SDK	3.0.67279
PM, BSC, UPC	Nanos++	0.7a (2013-04-22)
PM, BSC, UPC	Mercurium	1.99.0 (2013-05-07)
Open Source	ATLAS	3.10.1

Table 4.3: Third-party software used

4.1.4 Test framework

The tests were run using a python script, which in turn ran the individual compiled applications, providing appropriate runtime flags and environment variables. The results were presented on screen, and were in addition inserted into a database for further use. Sqlite was chosen as the database

Compiler	Optimization	Flags
gcc	Any	-std=gnu99 -Wall -pedantic -O3 -m64 -lOpenCL -lpapi
mcc	None	-std=gnu99 -Wall –ompss -O3 -m64 -lpapi
mcc	SSE	-std=gnu99 -Wall –ompss -O3 -m64 -lpapi -march=corei7 -flax-vector-conversions
mcc	AVX	-std=gnu99 -Wall –ompss -O3 -m64 -lpapi -march=corei7-avx -flax-vector-conversions

Table 4.4: Compiler flags used

engine for the results. The results were then extracted from the database, and output to figures using the python library matplotlib [1], and to tables using Jinja2 [2].

4.2 Experiment methodology

4.2.1 Performance counters

Before each experiment was run, the PAPI library was initialized using the code presented in section 3.1. As explained there, the wrap-around time of the RAPL counters proved somewhat unreliable. Therefore, the few times the counters wrapped around during benchmarking, the relevant test was simply run again. After running, the measurements are stopped, and the counters printed to stdout.

To ensure a fair benchmarking, each implementation is timed and measured from the same point in the application. This means that any time spent setting up the environment, (e.g. OpenCL context) or copying data is also included in the measurements.

4.2.2 Experiments

Each application is run 10 times for each configuration of problem size and optimizations. The median of the measurements was used in the results.

The Ivy Bridge processors come with dynamic frequency scaling, or CPU throttling. To ensure stable benchmarking results, the CPU clock speed was fixed to the maximum setting of 3.5 GHz.

4.2.3 Problem sizes

The problem sizes for the matrix multiplication are given as dimensions to each of the matrices. As described in section 2.4, all matrices in these experiments have the same dimensions.

The memory required for computing the result of two square matrices with single precision (4 byte) values is:

$$4bytes * 3 * N^2 = 12 * N^2 bytes \tag{4.1}$$

The problem sizes chosen for the experiments, along with their respective memory footprints are given in table 4.5. Not that for the simple OpenCL application, $N = 4096$ and $N = 8196$ were not run because they took too long.

N	Memory footprint
256	786 kB
512	3.1 MB
1024	12.6 MB
2048	50.3 MB
4096	201.3 MB
8192	805.3 MB

Table 4.5: Problem sizes and memory footprints

4.2.4 Defining flop counts

To calculate the performance of a system in FLOPS, it must be defined how many floating point operations is used for computing the correct result of a given input. It is possible to simply count the number of operations performed by the application using performance counters of the CPU. However, this would also count unnecessary operations due to a suboptimal algorithm, and therefore not be suitable for comparisons between different implementations.

Another definition of the flop count would be the number of "useful" floating point operations the algorithm performs, or put differently, how many operations are actually needed to produce a correct answer to the given problem. The formula for computing a matrix multiplication is seen in equation 2.3 on page 12. From this we can deduce the number of necessary floating point operations:

$$flops = 2 * N^3 \tag{4.2}$$

This is the flop count which will be used for performance measurements in this thesis. Integer operations and comparisons are ignored.

4.2.5 Measurement metrics

The recorded metrics are as follows:

1. $t = Time\ spent$ (measured in seconds).
2. $e = Energy\ used$ (measured in joules).

From these we calculate and compare the following:
MFLOPS, million floating point operations per second:

$$MFLOPS = \frac{flops}{second} = \frac{2 * N^3}{t * 10^6} \tag{4.3}$$

MFLOPS/W, million floating point operations per second per watt. This is simplified, as described in equation 2.4:

$$MFLOPS/W = \frac{flops}{joule} = \frac{2 * N^3}{e * 10^6} \tag{4.4}$$

Power dissipation, the rate of which energy is used, measured in watts.

$$Power = \frac{joules}{second} = \frac{e}{t} \tag{4.5}$$

Normalized energy delay product. The energy delay product is described in section 2.6.2. To be able to show the products for the different implementations in one graph, it is normalized against the non-optimized (not vectorized or single-threaded) version of the code.

$$EDP_{normalized} = \frac{e * t}{EDP_{non-optimized}} \tag{4.6}$$

Normalized energy delay squared. This is also explained in section 2.6.2, and as the previous metric, normalized.

$$EDD_{normalized} = \frac{e * t^2}{EDD_{non-optimized}} \tag{4.7}$$

Cache miss rate

$$Miss\ rate = \frac{Cache\ misses}{Cache\ accesses} \tag{4.8}$$

Chapter 5

Results and Discussion

In this chapter we will look at the performance results for the matrix multiplication kernel. The following sections present the results of the OpenCL versions and the OmpSs versions. The results are shown with different implementations, problem sizes, and optimization options.

5.1 Simple OpenCL implementation

This section will present the results for performance, energy, energy efficiency and cache miss rate for the simple (not transposed) OpenCL version of the kernel. The different optimizations shown for the OpenCL versions are different vecorization settings.

5.1.1 Performance

The performance, measured in MFLOPS, is presented in figure 5.1. As expected, the performance of the small problem sizes are quite low. This is mainly because the majority of the time spent is used on setting up the OpenCL context and environment. The time spent with setup and teardown become smaller compared to the computation time spent when the problem size increases, when we see the performance tapers off. Interest-

ingly, there is negligible difference between the different vectorization options, except the one using automatic vectorization. This might be because of low cache reuse of the second matrix, leading to the processor waiting for more data most of the time. As discussed in section 2.4, the matrices are stored in row-major order, but the second matrix in the multiplication is accessed in column-major order.

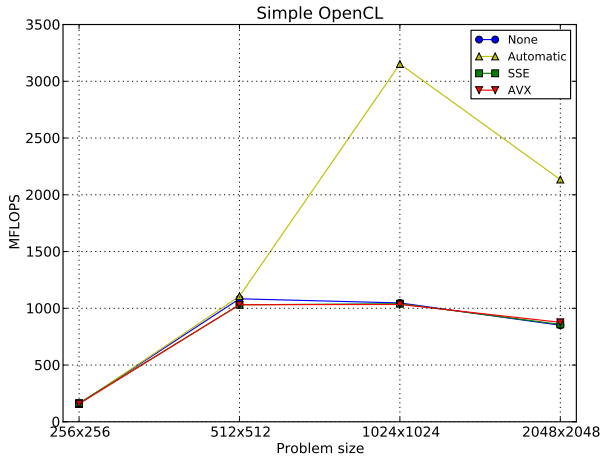
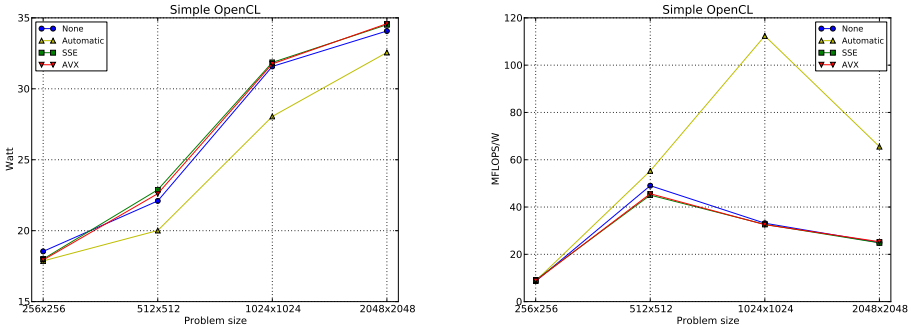


Figure 5.1: Performance for simple OpenCL matrix multiplication

The relatively high performance of the automatically vectorized version is hard to explain, as I do now know what the compiler has done to achieve this. As talked about in section 3.2, I do not have access to the generated assembly code. The performance decreases after $N = 1024$, most likely because the problem no longer fits in the level 3 processor cache (also seen in figure 5.4b).

5.1.2 Energy efficiency

Figure 5.2a shows the power dissipation during execution of the various tests. With the exception of the automatically vectorized version, they use energy at approximately the same rate. This shows that a vector operation and a scalar operation use about the same energy on this processor.



(a) Power dissipation

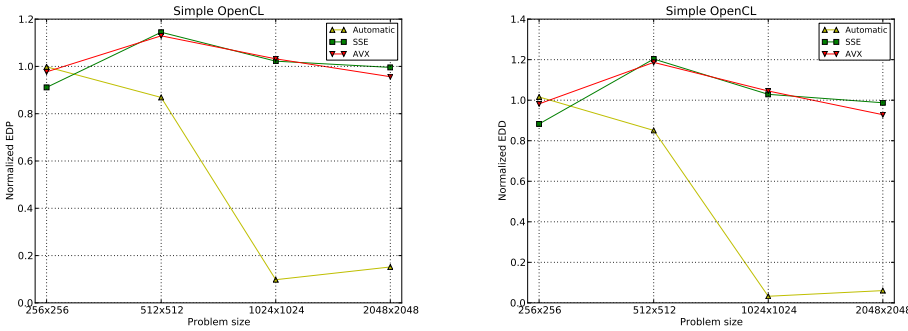
(b) Energy efficiency

Figure 5.2: Power dissipation and energy efficiency

The energy efficiency in figure 5.2b again shows that the automatically vectorized version performs the best. The other vectorization options perform equally well, peaking at the problem size 512×512.

5.1.3 Energy-delay products

Figure 5.3 shows the normalized energy delay products. It is normalized against the non-vectorized version, so as to be able to show them all in the same graphs. It is clear in both graphs that the automatically vectorized version is a lot more energy efficient. The other options are almost equal regarding the energy delay products. Not pictured is the non-optimized version, which the others are normalized against.



(a) Energy delay product

(b) Energy delay squared

Figure 5.3: Energy delay products

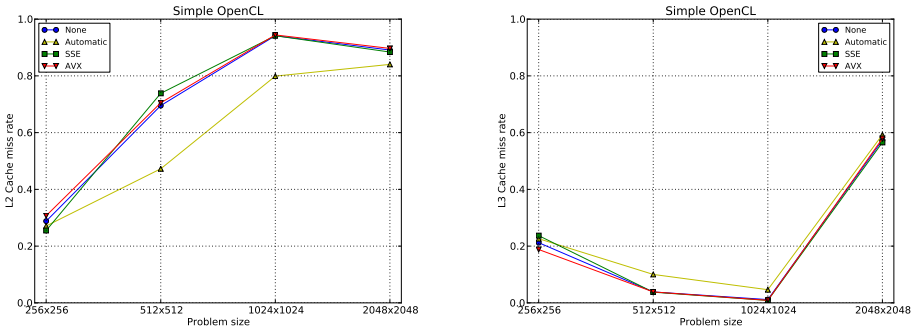
5.1.4 Cache miss rates

The cache miss rate for the level 2 cache is shown in figure 5.4a. Considering the size of the problems in matrix multiplication, we see that the level 2 cache quickly becomes almost useless. This is also probably in large part due to the poor cache reuse aspect of the algorithm in use.

Figure 5.4b shows the miss rate for the level 3 cache. As shown in equation 4.1, the memory required is $12 * N^3$. This means that the level 3 cache, which is as described in table 4.2, 8MB, will hold the entire problem at sizes below $N = 816$. We can however, see from the miss rate that even $N = 1024$ works quite well in the level 3 cache.

5.2 Transposed OpenCL implementation

This section will present the results for performance, energy, energy efficiency and cache miss rate for the transposed OpenCL version of the kernel. The different optimizations are here, as with the simple version, different vecorization settings.



(a) L2 cache miss rate

(b) L3 cache miss rate

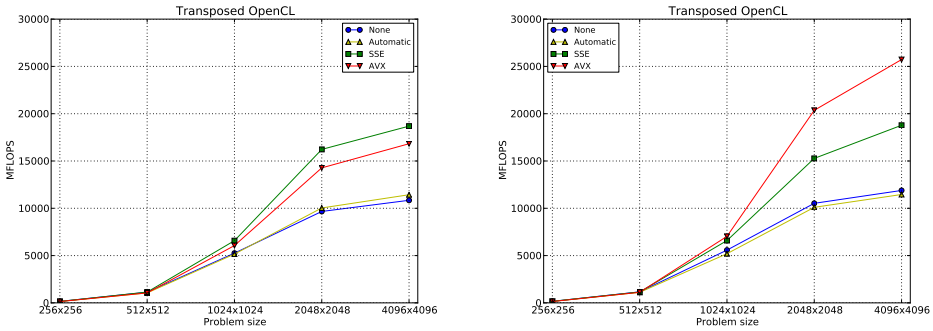
Figure 5.4: Cache miss rates

5.2.1 OpenCL versions

During the writing of this thesis, a new version of the OpenCL libraries was released by Intel. The first version used, 3.0.56860, was replaced by 3.0.67279 after the initial benchmarking was done. The differences in results are briefly presented here.

The old version from figure 5.5a shows that the SSE kernel outperforms the AVX kernel by 10%. As seen in section 3.2, there is no AVX version of the dot product, which is heavily used in this matrix multiplication implementation. Instead the function shown in listing 3.3 is used, which is simply two SSE instructions serially executed. This would explain the situation if they were almost equal, but it is strange that the AVX version performs worse.

Figure 5.5b shows the performance of the new version. We can see that the non-vectorized, automatic, and SSE versions perform similarly, but the kernel employing AVX is much improved. There still is no `float8` dot product in the new version of the OpenCL implementation, but it is assumed that the compiler combines the two `float4` dot products to an AVX instruction.



(a) Old version

(b) New version

Figure 5.5: Transposed OpenCL performance, different versions

5.2.2 Performance

Here we see from figure 5.6 a lot better performance than the simple version. This time the automatic vectorization could not do much for the performance, which might be because of the complexity of the algorithm compared to the simple version. Both SSE and AVX improve the performance by a lot, but far from the theoritical speedup of 4 and 8, respectively.

5.2.3 Energy efficiency

From figure 5.7a we see that all the vecorization options use energy at about the same rate. As discussed in the previous section, this shows that vector operations and scalar operations use approximately the same amount of energy. As with the simple version, the automatically vectorized option is the outlier, this time using slightly more energy per second.

The measured energy efficiency in MFLOPS/W is shown in figure 5.7b. As expected, the efficiency of the smaller problem sizes is quite low, due to the relatively large part of the measure time is spent setting up the OpenCL environment. After reaching 2048×2048, the efficiency almost completely

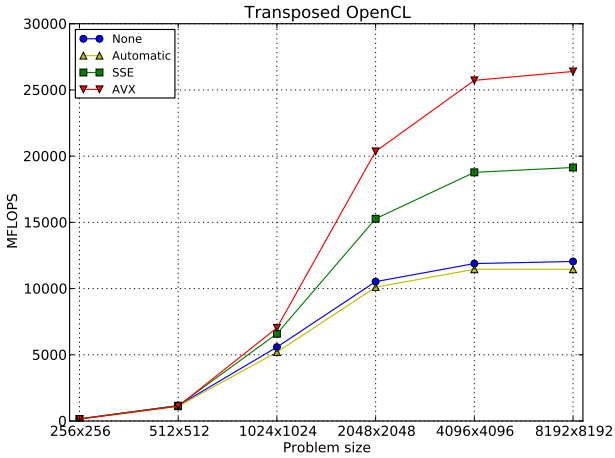


Figure 5.6: Performance for transposed OpenCL matrix multiplication

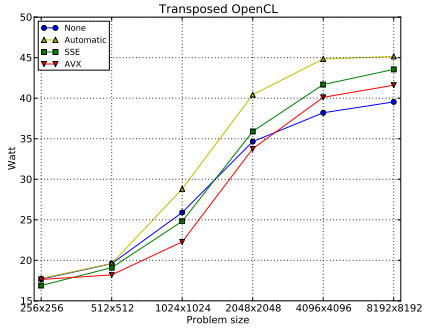
flattens, which is again most likely because of now negligible time spent setting up OpenCL. The AVX version is again the winner, being most efficient for all problem sizes.

5.2.4 Energy-delay products

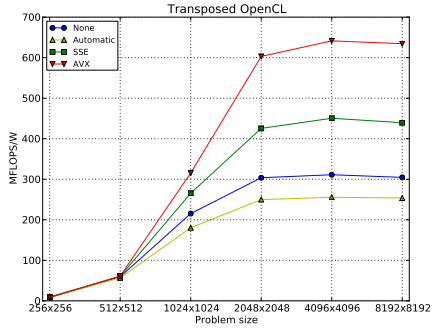
Figure 5.8 shows the energy delay products. Here we also see that the AVX version is most efficient for all problem sizes. The automatically vectorized version is slightly worse than the plain version for all sizes. The numbers are normalized against the non-optimized version, which is not shown.

5.2.5 Cache miss rates

Here we see from figure 5.9 part of the reason this implementation is so much stronger than the simple version. The miss rate for both the level 2 and the level 3 caches are much lower than in section 5.1. They are approximately equal for the smaller sizes on both levels, diverging slightly

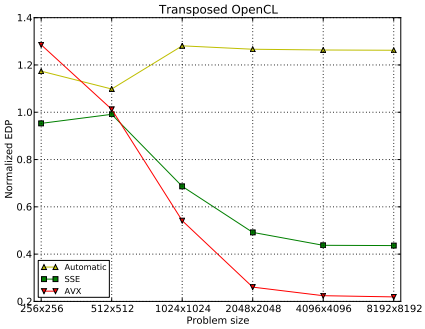


(a) Power dissipation

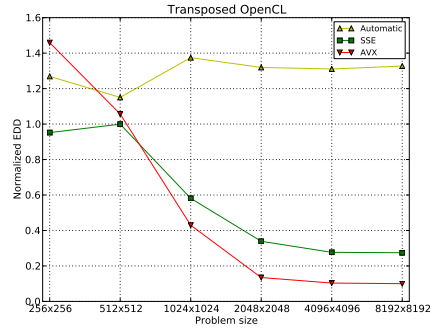


(b) Energy efficiency

Figure 5.7: Power dissipation and energy efficiency, transposed OpenCL



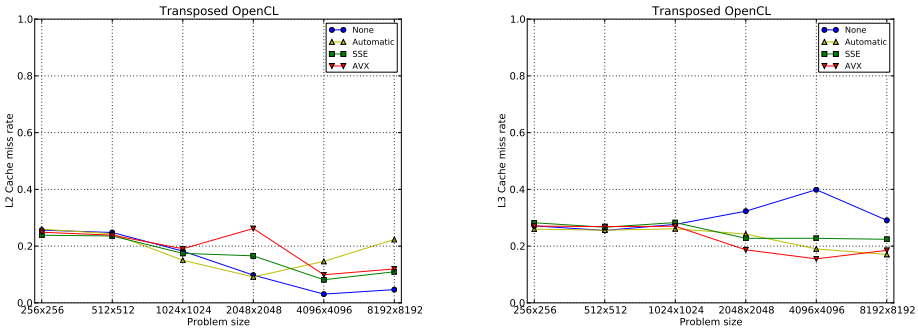
(a) Energy delay product



(b) Energy delay squared

Figure 5.8: Energy delay products

as the size increases. There are only small differences between the various optimizations.



(a) L2 cache miss rate

(b) L3 cache miss rate

Figure 5.9: Cache miss rates

5.3 OmpSs

This section will present and discuss the performance and energy results for the OmpSs implementation of the matrix multiplication developed by me. The optimizations available to this version are vectorization and number of threads. One, four, and eight threads were tested.

5.3.1 Performance

From figure 5.10 we see the performance of the application. The effects of both multi-threading and vectorization are as expected, with the eight-threaded AVX version being the fastest. We also see that hyper-threading is quite effective, increasing the performance for all vectorization options.

Vectorization is also shown to affect the performance positively. Regardless of the number of threads, it increases the performance significantly.

5.3.2 Energy dfficiency

Figure 5.11a shows the effect of multi-threading on power spent. Four threads is shown to use on average 107% more energy per second than

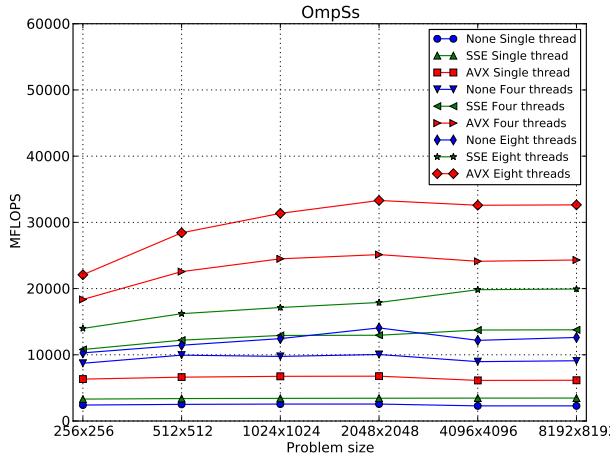


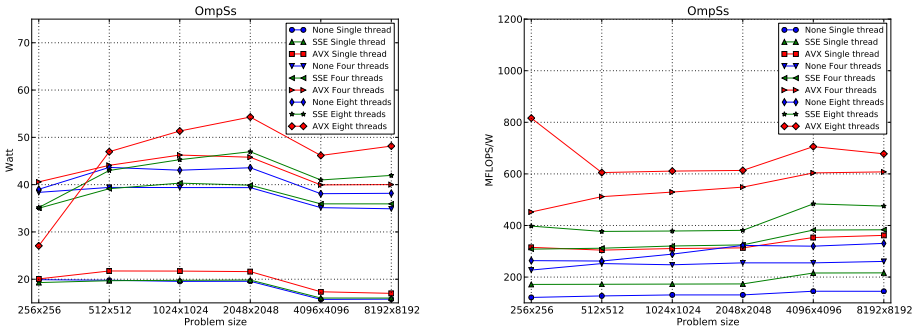
Figure 5.10: Performance for OmpSs matrix multiplication

the single-threaded application, while eight threads use 127% more. The vectorized code also seems to spend energy somewhat faster, contradicting what we saw in section 5.1 and 5.2.

Even though both multi-threading and vectorization consume energy faster, we see from figure 5.11b that it is worth it with regards to efficiency. Both optimizations produce greater MFLOPS/W than the corresponding non-optimized applications.

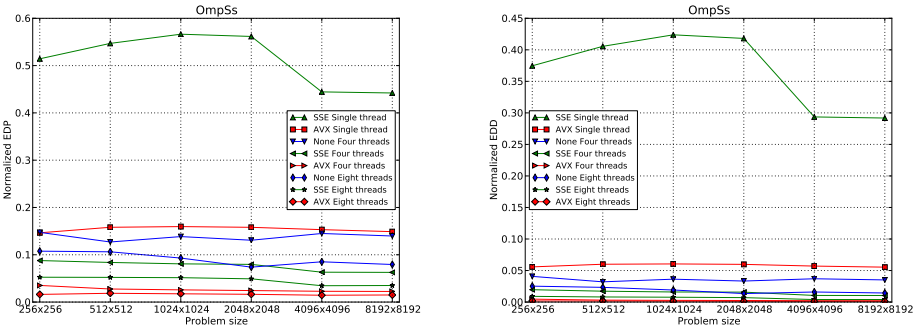
5.3.3 Energy delay products

Looking at both energy delay products in figure 5.12 it looks like the single-threaded SSE application stands out. On closer inspection we see that it is actually all the single-threaded versions standing out. The other combinations are all quite stable and low, with the eight-threaded AVX version again being the winner. Note that the numbers are normalized against the non-optimized version, which is not shown.



(a) Power dissipation (b) Energy efficiency

Figure 5.11: Power dissipation and energy efficiency, OmpSs

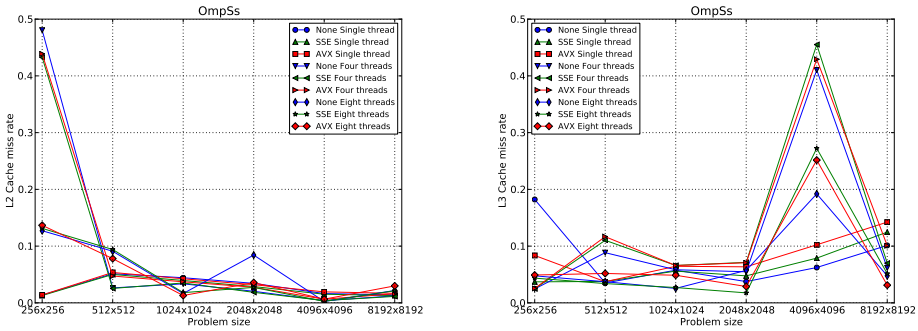


(a) Energy delay product (b) Energy delay squared

Figure 5.12: Energy delay products

5.3.4 Cache miss rate

Figure 5.13a shows that the miss rate for the level 2 cache is, for all problem sizes above $N = 256$, very low. This is due to the fact that one sub-block multiplication always fits inside the level 2 cache.



(a) L2 cache miss rate

(b) L3 cache miss rate

Figure 5.13: Cache miss rates

From figure 5.13b there are more interesting results. We see that the number of threads is the factor determining the level 3 cache miss rate, with the single-threaded, four-threaded, and eight-threaded runs being quite similar, with the four-threaded being the worst. There is not much difference regarding the vectorization. There is a big spike in the miss rate at $N = 4096$ for four and eight threads. This is probably because of the block size chosen for this problem size being too big for the L3 cache. However, even with the poor level 3 miss rate, this block size performed the best (see section 3.3.1).

Figure 5.14 shows the number of accesses at the level 2 cache steadily rising for all optimizations. This metric also represents the number of misses at the level 1 cache, as any miss there would necessarily also mean an access at the next level. We cannot see the level one miss rate, as there are no counters for the accesses at that level, but this metric gives us an idea of how well the level one cache is used. Note the logarithmic scale. We clearly see from the figure three groups, formed from the single-threaded, four-threaded, and eight-threaded applications. As expected, vectorization has no impact on the number of level 2 cache accesses.

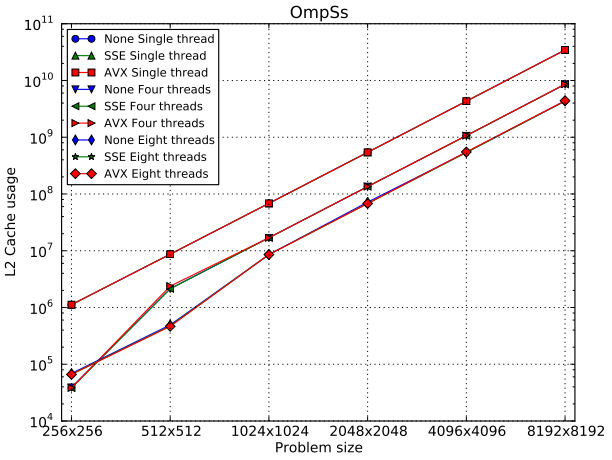


Figure 5.14: Level 2 cache usage, OmpSs

5.4 OmpSs from BSC

This section will present the results from the OmpSs implementation provided by BSC, which uses ATLAS for the actual computation. As ATLAS is already AVX vectorized, the only optimization available is the number of threads. Again, one, four, and eight threads were tested.

5.4.1 Performance

We can see from figure 5.15 a huge improvement from the previous applications. We also see that hyper-threading has a small but constant negative impact on performance. The effect of multi-threading is quite big, with four threads having 84% higher performance than the single-threaded application. The negative impact of hyper-threading is most likely due to pairs of threads sharing a cache. This means that we get lower cache reuse, and lower performance. Additionally, because parallelization is introduced outside the ATLAS library, it will wrongly assume each thread has access

to the full 32kB of level 1 cache.

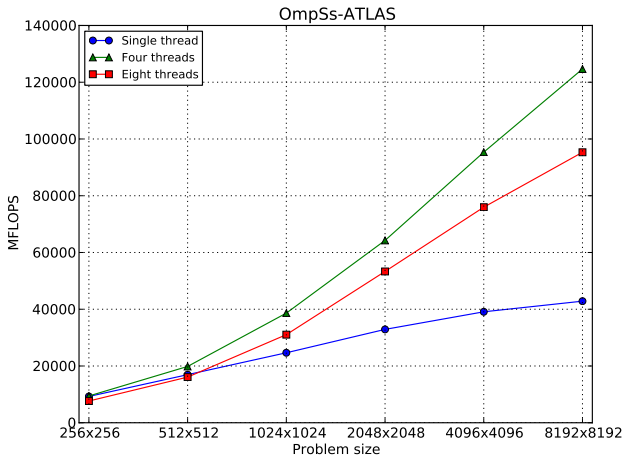
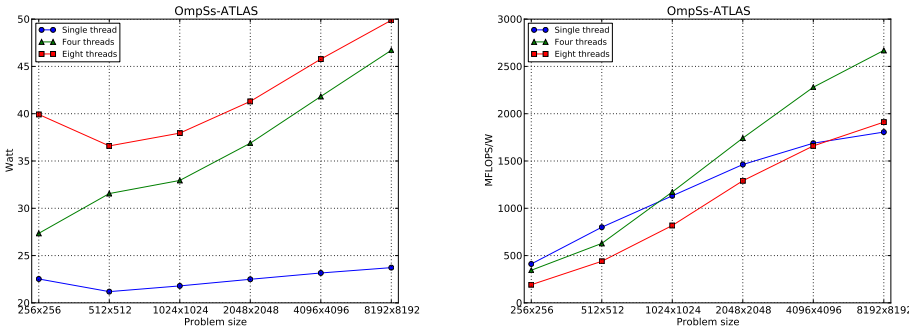


Figure 5.15: Performance for OmpSs BSC matrix multiplication

5.4.2 Energy dfficiency

Figure 5.16a shows that using eight threads spends energy faster than four threads. Interestingly, the single-threaded power usage is almost unaffected by problem size, whereas the four-threaded power usage increases steadily. At the biggest problem size the four-threaded power spent is 96% higher than the single-threaded.

After seeing both the performance and the power usage of four threads versus eight threads, the result of figure 5.16b is not surprising. It shows that the energy efficiency of four threads is quite a bit better than eight threads. The single-threaded version performs remarkably well, being passed by the four-threaded at $N = 1024$, and the eight-threaded at $N = 8192$.



(a) Power dissipation (b) Energy efficiency

Figure 5.16: Power dissipation and energy efficiency, OmpSs BSC

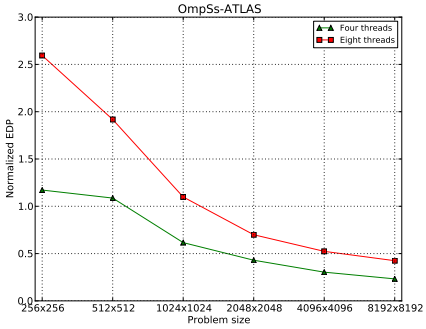
5.4.3 Energy delay products

The energy delay products in figure 5.17 also show that four threads are more efficient than eight, with the difference decreasing as the problem size increases. When looking at the EDP, they are both better than the single-threaded application after 1024×1024. Using the EDD, which favors performance, they cross the single-threaded somewhat earlier, with both being more efficient after 512×512. The numbers are normalized against the single-threaded version, which is not shown.

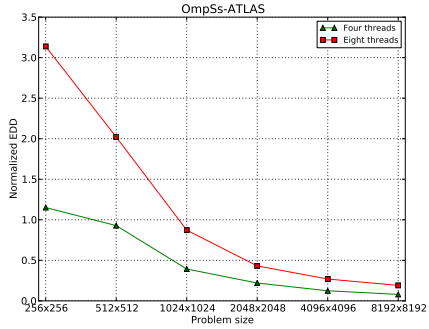
5.4.4 Cache miss rate

Figure 5.18a shows the miss rate for the level 2 cache. It is generally quite low, with not much difference between the number of threads. The ATLAS library should, and apparantly does, divide the work into smaller tasks which fit nicely into the level 2 cache.

The miss rate for the level 3 cache is shown in figure 5.18b. The single-threaded application increases sharply at $N = 1024$, which is the first problem size which does not fit into the level 3 cache. The four-threaded

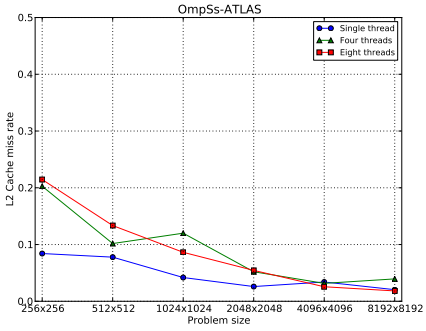


(a) Energy delay product

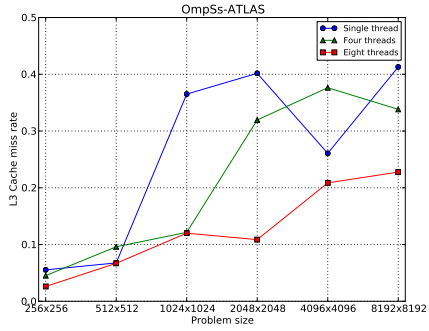


(b) Energy delay squared

Figure 5.17: Energy delay products



(a) L2 cache miss rate



(b) L3 cache miss rate

Figure 5.18: Cache miss rates

version however, increases similarly only at $N = 2048$. The eight-threaded version steadily increases with the problem size.

Figure 5.19 shows the number of level 2 cache accesses. Note the logarithmic scale, and the fact that this number also represents the level 1

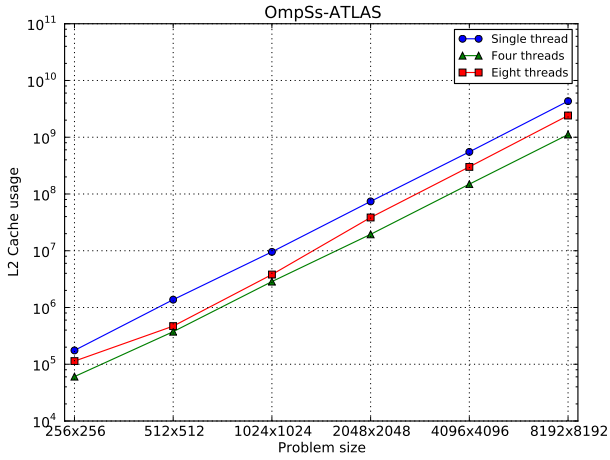


Figure 5.19: Level 2 cache usage, OmpSs BSC

cache misses, as explained in section 5.3.4. All optimizations are steadily increasing, as expected. The single-threaded version misses the most, followed by the eight-threaded, with the four-threaded performing the best. This correlates with the performance and energy results we saw earlier.

Chapter 6

Conclusion

Four different implementations of matrix multiplication have been run, measured, and analyzed. The performance and energy efficiency of the different versions have been compared and discussed. This section will conclude the report.

6.1 OpenCL

Two OpenCL implementations were tested. Here we look at the differences between them, and on the effect of vectorization.

6.1.1 Transposition

The effects of transposing the second matrix were quite big. For the problem size $N = 2048$, the increase in MFLOPS was 1354%, with the largest increase in the AVX version. The increase was also evident in the energy efficiency. With almost no increase in power, the increase in MFLOPS/W were nearly equal to the increase in MFLOPS (1328% for $N = 2048$). Some of the reason for the increase in performance is shown in the cache miss rates, with both the L2 and the L3 cache being much better utilized in the transposed version.

6.1.2 Vectorization

Vectorization also had a big impact on performance. Looking at the transposed implementation, the increase in performance between the non-vectorized and the SSE version was 58% for the big problem sizes. AVX was quite a bit faster than SSE, being 118% faster than the non-vectorized code. The AVX version was also the most energy efficient. It used only 5% more power, meaning that it gave 107% more MFLOPS/W than the non-vectorized version. There was almost no difference in the cache miss rate, which is to be expected.

6.2 OmpSs

Two OmpSs implementions were tested. Here we look at the difference between between them, and on the effect of vectorization, scheduling, and multi-threading.

6.2.1 ATLAS

The OmpSs application provided by BSC generally performed much better than the application developed during this thesis. At the biggest problem size tested, the performance was 282% better. It was only beat at the smallest problem sizes of $N = 256$ and $N = 512$. The difference in energy efficiency (MFLOPS/W) was even bigger, peaking at 294%.

This difference probably comes from many reasons, the one seen in this thesis was a big improvement in cache usage. The ATLAS application uses the level 1 cache much better, with almost an order of magnitude less level 2 accesses. They were similar in the level 2 miss rate, with level 3 varying widely.

6.2.2 Vectorization

The OmpSs implementation from BSC was already vectorized in ATLAS. From the results of the application developed during this thesis, we see that

vectorization helps quite a bit, although far from the theoretical maximum of 4 and 8 times speedup for SSE and AVX respectively. AVX performed on average 152% better than the non-vectorized, while SSE performed 40% better.

The vectorized code used somewhat more power, but the performance gain outweighed it, making the energy efficiency (MFLOPS/W) 130% better for the AVX optimized code than the non-optimized.

6.2.3 Multi-threading

The multi-threaded runs performed as expected much better than the single-threaded. Also here it was far from the theoretical gain, with the four-threaded being 84% faster than the single-threaded for the BSC version, and hyper-threading being 393% faster for my version. When using ATLAS, hyper-threading produced worse results than four threads.

While spending significanly more energy than a single-threaded application, the performance gains were again worth it. The BSC version was 11% more efficient running four threads, while my version was 117% more efficient running eight measured in MFLOPS/W.

6.3 Further work

In this section, possible further work is presented.

6.3.1 OpenCL kernels in OmpSs

Being able to run OpenCL kernels from OmpSs is essential if one wishes to utilize a GPU not from NVIDIA. The embedded GPU on the Intel Ivy Bridge is running idle during all the benchmarking in this thesis, utilizing it would likely increase performance quite a bit.

6.3.2 Arndale OpenCL

Running the tests on the Arndale development board with its Exynos 5 chip would be quite interesting, especially because it has been chosen to be used in the Mont Blanc project. There were no OpenCL drivers released for the GPU or the CPU at the time of writing.

6.3.3 GPU testing

The Intel Ivy Bridge has an embedded GPU, the Intel HD Graphics 4000, which would be very interesting to run the tests on. One could also try to split the problems between the CPU and GPU, and compare the performance and energy results. At the time of writing, there was no OpenCL driver for the GPU released for Linux.

6.3.4 Complete system energy measurements

This thesis only measured the energy usage of the CPU. The CPU is one of the biggest energy users in the system, but many other factors play a role. Measuring the entire system would give a more complete view of the energy usage of a solution.

6.3.5 Additional applications

Only the matrix multiplication kernel was tested in this thesis. More kernels should be tested to find out more about performance versus energy usage.

6.4 Concluding remarks

In this thesis we have seen the big difference a tuned implementation of an algorithm can have, both on performance and energy efficiency. The two OmpSs applications scored better than both the OpenCL implementations in both regards. Both vectorization and multi-threading have very positive results especially on performance, but also energy efficiency. Hyper-threading has shown mixed results.

Appendices

Appendix A

Tabulated Data

This appendix contains the complete data set from the experiments whose results are shown in section 5.

Simple OpenCL - MFLOPS				
N	**None**	**Automatic**	**SSE**	**AVX**
256	160	157	165	159
512	1084	1106	1031	1031
1024	1047	3152	1040	1034
2048	851	2134	858	876

Table A.1: MFLOPS, Simple OpenCL

Simple OpenCL - Watt				
N	**None**	**Automatic**	**SSE**	**AVX**
256	19	18	18	18
512	22	20	23	23
1024	32	28	32	32
2048	34	33	35	35

Table A.2: Watt, Simple OpenCL

Simple OpenCL - MFLOPS/W				
N	**None**	**Automatic**	**SSE**	**AVX**
256	9	9	9	9
512	49	55	45	46
1024	33	112	33	33
2048	25	66	25	25

Table A.3: MFLOPS/W, Simple OpenCL

Simple OpenCL - Normalized EDP				
N	**None**	**Automatic**	**SSE**	**AVX**
256	1.0	0.998	0.911	0.977
512	1.0	0.869	1.144	1.129
1024	1.0	0.098	1.022	1.032
2048	1.0	0.152	0.996	0.956

Table A.4: Normalized EDP, Simple OpenCL

Simple OpenCL - Normalized EDD				
N	**None**	**Automatic**	**SSE**	**AVX**
256	1.0	1.016	0.883	0.982
512	1.0	0.851	1.203	1.186
1024	1.0	0.033	1.029	1.045
2048	1.0	0.061	0.987	0.928

Table A.5: Normalized EDD, Simple OpenCL

Simple OpenCL - L2 Cache miss rate				
N	**None**	**Automatic**	**SSE**	**AVX**
256	0.288	0.271	0.255	0.307
512	0.695	0.473	0.738	0.705
1024	0.942	0.799	0.941	0.944
2048	0.891	0.84	0.884	0.897

Table A.6: L2 Cache miss rate, Simple OpenCL

Simple OpenCL - L3 Cache miss rate				
N	**None**	**Automatic**	**SSE**	**AVX**
256	0.212	0.226	0.237	0.188
512	0.039	0.1	0.038	0.039
1024	0.012	0.046	0.009	0.009
2048	0.573	0.593	0.565	0.579

Table A.7: L3 Cache miss rate, Simple OpenCL

Transposed OpenCL - MFLOPS				
N	**None**	**Automatic**	**SSE**	**AVX**
256	163	150	163	143
512	1166	1113	1156	1117
1024	5579	5199	6589	7032
2048	10525	10103	15274	20359
4096	11887	11458	18777	25724
8192	12046	11458	19143	26396

Table A.8: MFLOPS, Transposed OpenCL

Transposed OpenCL - Watt				
N	**None**	**Automatic**	**SSE**	**AVX**
256	18	18	17	18
512	20	20	19	18
1024	26	29	25	22
2048	35	40	36	34
4096	38	45	42	40
8192	40	45	44	42

Table A.9: Watt, Transposed OpenCL

Transposed OpenCL - MFLOPS/W				
N	**None**	**Automatic**	**SSE**	**AVX**
256	9	8	10	8
512	60	57	61	61
1024	215	180	266	316
2048	304	250	426	603
4096	311	256	450	641
8192	305	254	440	634

Table A.10: MFLOPS/W, Transposed OpenCL

Transposed OpenCL - Normalized EDP				
N	**None**	**Automatic**	**SSE**	**AVX**
256	1.0	1.174	0.953	1.285
512	1.0	1.098	0.991	1.012
1024	1.0	1.281	0.687	0.542
2048	1.0	1.266	0.492	0.26
4096	1.0	1.263	0.437	0.224
8192	1.0	1.262	0.436	0.219

Table A.11: Normalized EDP, Transposed OpenCL

Transposed OpenCL - Normalized EDD				
N	**None**	**Automatic**	**SSE**	**AVX**
256	1.0	1.269	0.952	1.459
512	1.0	1.15	0.999	1.057
1024	1.0	1.374	0.581	0.43
2048	1.0	1.319	0.339	0.135
4096	1.0	1.311	0.277	0.104
8192	1.0	1.327	0.274	0.1

Table A.12: Normalized EDD, Transposed OpenCL

Transposed OpenCL - L2 Cache miss rate				
N	**None**	**Automatic**	**SSE**	**AVX**
256	0.256	0.26	0.238	0.249
512	0.248	0.244	0.236	0.239
1024	0.182	0.15	0.175	0.191
2048	0.098	0.092	0.166	0.262
4096	0.031	0.146	0.081	0.099
8192	0.047	0.223	0.11	0.12

Table A.13: L2 Cache miss rate, Transposed OpenCL

Transposed OpenCL - L3 Cache miss rate				
N	**None**	**Automatic**	**SSE**	**AVX**
256	0.271	0.26	0.282	0.271
512	0.255	0.256	0.267	0.269
1024	0.277	0.261	0.283	0.27
2048	0.323	0.243	0.227	0.187
4096	0.399	0.19	0.228	0.155
8192	0.291	0.17	0.224	0.185

Table A.14: L3 Cache miss rate, Transposed OpenCL

OmpSs - MFLOPS Eight threads			
N	**None**	**SSE**	**AVX**
256	10290	13981	22090
512	11444	16208	28421
1024	12443	17141	31360
2048	14060	17897	33306
4096	12180	19832	32584
8192	12618	19932	32640

Table A.15: MFLOPS, OmpSs

OmpSs - Watt Eight threads			
N	**None**	**SSE**	**AVX**
256	39	35	27
512	44	43	47
1024	43	45	51
2048	44	47	54
4096	38	41	46
8192	38	42	48

Table A.16: Watt, OmpSs

OmpSs - MFLOPS/W Eight threads			
N	**None**	**SSE**	**AVX**
256	264	398	816
512	262	377	605
1024	289	379	611
2048	323	381	613
4096	320	484	706
8192	331	475	678

Table A.17: MFLOPS/W, OmpSs

OmpSs - Normalized EDP Eight threads			
N	**None**	**SSE**	**AVX**
256	0.108	0.053	0.016
512	0.106	0.052	0.019
1024	0.093	0.052	0.017
2048	0.074	0.049	0.016
4096	0.085	0.035	0.014
8192	0.079	0.035	0.015

Table A.18: Normalized EDP, OmpSs

OmpSs - Normalized EDD Eight threads			
N	**None**	**SSE**	**AVX**
256	0.025	0.009	0.002
512	0.023	0.008	0.002
1024	0.019	0.008	0.001
2048	0.013	0.007	0.001
4096	0.016	0.004	0.001
8192	0.014	0.004	0.001

Table A.19: Normalized EDD, OmpSs

OmpSs - L2 Cache miss rate Eight threads			
N	**None**	**SSE**	**AVX**
256	0.127	0.131	0.137
512	0.091	0.094	0.078
1024	0.017	0.018	0.013
2048	0.084	0.027	0.036
4096	0.002	0.003	0.007
8192	0.022	0.021	0.03

Table A.20: L2 Cache miss rate, OmpSs

OmpSs - L3 Cache miss rate Eight threads			
N	**None**	**SSE**	**AVX**
256	0.048	0.044	0.049
512	0.037	0.034	0.052
1024	0.025	0.027	0.049
2048	0.057	0.018	0.029
4096	0.192	0.272	0.252
8192	0.048	0.051	0.031

Table A.21: L3 Cache miss rate, OmpSs

OmpSs - L2 Cache usage Eight threads			
N	**None**	**SSE**	**AVX**
256	$6.88 * 10^4$	$6.67 * 10^4$	$6.61 * 10^4$
512	$4.93 * 10^5$	$4.77 * 10^5$	$4.66 * 10^5$
1024	$8.55 * 10^6$	$8.57 * 10^6$	$8.57 * 10^6$
2048	$7.18 * 10^7$	$6.78 * 10^7$	$6.79 * 10^7$
4096	$5.41 * 10^8$	$5.37 * 10^8$	$5.51 * 10^8$
8192	$4.34 * 10^9$	$4.33 * 10^9$	$4.4 * 10^9$

Table A.22: L2 Cache usage, OmpSs

OmpSs-ATLAS - MFLOPS			
N	**Single thread**	**Four threads**	**Eight threads**
256	9267	9433	7659
512	16949	19840	16085
1024	24660	38633	31046
2048	32900	64254	53314
4096	39098	95348	75959
8192	42845	124656	95315

Table A.23: MFLOPS, OmpSs-ATLAS

OmpSs-ATLAS - Watt			
N	**Single thread**	**Four threads**	**Eight threads**
256	23	27	40
512	21	32	37
1024	22	33	38
2048	22	37	41
4096	23	42	46
8192	24	47	50

Table A.24: Watt, OmpSs-ATLAS

OmpSs-ATLAS - MFLOPS/W			
N	**Single thread**	**Four threads**	**Eight threads**
256	411	345	192
512	800	629	440
1024	1132	1173	818
2048	1463	1742	1291
4096	1688	2280	1659
8192	1806	2669	1911

Table A.25: MFLOPS/W, OmpSs-ATLAS

OmpSs-ATLAS - Normalized EDP			
N	**Single thread**	**Four threads**	**Eight threads**
256	1.0	1.172	2.594
512	1.0	1.087	1.918
1024	1.0	0.616	1.099
2048	1.0	0.43	0.699
4096	1.0	0.304	0.524
8192	1.0	0.233	0.425

Table A.26: Normalized EDP, OmpSs-ATLAS

OmpSs-ATLAS - Normalized EDD			
N	**Single thread**	**Four threads**	**Eight threads**
256	1.0	1.151	3.139
512	1.0	0.929	2.021
1024	1.0	0.393	0.873
2048	1.0	0.22	0.431
4096	1.0	0.124	0.269
8192	1.0	0.08	0.191

Table A.27: Normalized EDD, OmpSs-ATLAS

OmpSs-ATLAS - L2 Cache miss rate			
N	**Single thread**	**Four threads**	**Eight threads**
256	0.084	0.203	0.215
512	0.078	0.102	0.133
1024	0.042	0.12	0.086
2048	0.026	0.052	0.054
4096	0.034	0.032	0.026
8192	0.02	0.039	0.018

Table A.28: L2 Cache miss rate, OmpSs-ATLAS

OmpSs-ATLAS - L3 Cache miss rate			
N	**Single thread**	**Four threads**	**Eight threads**
256	0.055	0.045	0.026
512	0.068	0.096	0.067
1024	0.365	0.122	0.12
2048	0.402	0.319	0.109
4096	0.261	0.376	0.209
8192	0.413	0.338	0.228

Table A.29: L3 Cache miss rate, OmpSs-ATLAS

OmpSs-ATLAS - L2 Cache usage			
N	**Single thread**	**Four threads**	**Eight threads**
256	$1.75 * 10^5$	$6.06 * 10^4$	$1.13 * 10^5$
512	$1.37 * 10^6$	$3.75 * 10^5$	$4.7 * 10^5$
1024	$9.55 * 10^6$	$2.86 * 10^6$	$3.81 * 10^6$
2048	$7.4 * 10^7$	$1.95 * 10^7$	$3.85 * 10^7$
4096	$5.52 * 10^8$	$1.49 * 10^8$	$3.01 * 10^8$
8192	$4.32 * 10^9$	$1.11 * 10^9$	$2.41 * 10^9$

Table A.30: L2 Cache usage, OmpSs-ATLAS

Appendix B

OpenCL kernels

This appendix contains the kernels in use in the applications presented in section 3.2.

Listing B.1 Simple OpenCL

```
__kernel void matmul(__global float *a, __global float *b, __global float *c) {
    int n = get_global_size(0);

    int row = get_global_id(0);
    int col = get_global_id(1);

    float sum = 0.0f;
    for(int k = 0; k < n; k++) {
        sum += a[n*row + k] * b[n*k + col];
    }
    c[n*row + col] = sum;
}
```

Listing B.2 Simple OpenCL with SSE

```
__kernel __attribute__((vec_type_hint(float4)))
void matmul(__global float4 *a, __global float *b, __global float *c) {
    int n = get_global_size(0);
    int row = get_global_id(0);
    int col = get_global_id(1);

    float sum = 0.0f;
    for(int k = 0; k < n; k += 4) {
        float4 bvec = (float4) (b[n*k + col],
                                b[n*(k+1) + col],
                                b[n*(k+2) + col],
                                b[n*(k+3) + col]);
        sum += dot(a[(n*row + k)/4], bvec);
    }
    c[n*row + col] = sum;
}
```

Listing B.3 Simple OpenCL with AVX

```
__kernel __attribute__((vec_type_hint(float8)))
void matmul(__global float8 *a, __global float *b, __global float *c) {
    int n = get_global_size(0);
    int row = get_global_id(0);
    int col = get_global_id(1);

    float sum = 0.0f;
    for(int k = 0; k < n; k += 8) {
        float8 bvec = (float8) (b[n*(k+0) + col],
                                b[n*(k+1) + col],
                                b[n*(k+2) + col],
                                b[n*(k+3) + col],
                                b[n*(k+4) + col],
                                b[n*(k+5) + col],
                                b[n*(k+6) + col],
                                b[n*(k+7) + col]);
        sum += dot(a[(n*row+k)/8].lo, bvec.lo) + dot(a[(n*row+k)/8].hi, bvec.hi);
    }
    c[n*row + col] = sum;
}
```

Listing B.4 Transposed OpenCL, transposition

```
__kernel void transpose(__global float *g_mat) {
    uint size = SIZE;
    float loc;
    int col = get_global_id(0);
    int row = 0;
    while (col >= size) {
        col -= size--;
        row++;
    }
    col += row;
    size += row;

    if (row != col) {
        loc = g_mat[row * size + col];
        g_mat[row * size + col] = g_mat[col * size + row];
        g_mat[col * size + row] = loc;
    }
}
```

Listing B.5 Transposed OpenCL, multiplication

```
__kernel void mult(__global float *a, __global float *b, __global float *c) {
    int a_row = get_global_id(0);
    int b_row = get_global_id(1);
    int num_rows = get_global_size(0);
    int vectors_per_row = num_rows;

    a += a_row * vectors_per_row;
    b += b_row * vectors_per_row;

    float sum = 0.0f;
    for (int i = 0; i < vectors_per_row; i++) {
        sum += a[i] * b[i];
    }
    c[a_row * num_rows + b_row] = sum;
}
```

Listing B.6 Transposed OpenCL with SSE, transposition

```
__kernel __attribute__((vec_type_hint(float4)))
void transpose(__global float4 *g_mat) {
    uint size = SIZE;
    __global float4 *src, *dst;
    float4 l_mat[4];

    int col = get_global_id(0);
    int row = 0;
    while (col >= size) {
        col -= size--;
        row++;
    }
    col += row;
    size += row;

    src = g_mat + row * size * 4 + col;
    for (int i = 0; i < 4; i++)
        l_mat[i] = src[i*size];

    if (row == col) {
        src[0]      = (float4)(l_mat[0].x, l_mat[1].x, l_mat[2].x, l_mat[3].x);
        src[size]   = (float4)(l_mat[0].y, l_mat[1].y, l_mat[2].y, l_mat[3].y);
        src[2*size] = (float4)(l_mat[0].z, l_mat[1].z, l_mat[2].z, l_mat[3].z);
        src[3*size] = (float4)(l_mat[0].w, l_mat[1].w, l_mat[2].w, l_mat[3].w);
    } else {
        dst = g_mat + col * size * 4 + row;

        src[0]      = (float4)(dst[0].x, dst[size].x, dst[2*size].x, dst[3*size].x)
    ;
        src[size]   = (float4)(dst[0].y, dst[size].y, dst[2*size].y, dst[3*size].y)
    ;
        src[2*size] = (float4)(dst[0].z, dst[size].z, dst[2*size].z, dst[3*size].z)
    ;
        src[3*size] = (float4)(dst[0].w, dst[size].w, dst[2*size].w, dst[3*size].w)
    ;

        dst[0]      = (float4)(l_mat[0].x, l_mat[1].x, l_mat[2].x, l_mat[3].x);
        dst[size]   = (float4)(l_mat[0].y, l_mat[1].y, l_mat[2].y, l_mat[3].y);
        dst[2*size] = (float4)(l_mat[0].z, l_mat[1].z, l_mat[2].z, l_mat[3].z);
        dst[3*size] = (float4)(l_mat[0].w, l_mat[1].w, l_mat[2].w, l_mat[3].w);
    }
}
```

Listing B.7 Transposed OpenCL with SSE, multiplication

```
__kernel __attribute__((vec_type_hint(float4)))
void mult(__global float4 *a, __global float4 *b, __global float *c) {
    int a_row = get_global_id(0);
    int b_row = get_global_id(1);
    int num_rows = get_global_size(0);
    int vectors_per_row = num_rows / 4;

    a += a_row * vectors_per_row;
    b += b_row * vectors_per_row;

    float sum = 0.0f;
    for (int i = 0; i < vectors_per_row; i++) {
        sum += dot(a[i], b[i]);
    }
    c[a_row * num_rows + b_row] = sum;
}
```

Listing B.8 Transposed OpenCL with AVX, transposition

```
1  __kernel __attribute__((vec_type_hint(float8)))
2  void transpose(__global float8 *g_mat) {
3      uint size = SIZE;
4      __global float8 *src, *dst;
5      float8 l_mat[8];
6
7      uint col = get_global_id(0);
8      uint row = 0;
9      while (col >= size) {
10         col -= size--;
11         row++;
12     }
13     col += row;
14     size += row;
15
16     src = g_mat + row * size * 8 + col;
17     for (int i = 0; i < 8; i++)
18         l_mat[i] = src[i * size];
19
20     if (row == col) {
21         src[0]      = (float8) (l_mat[0].s0, l_mat[1].s0, l_mat[2].s0, l_mat[3].s0,
       l_mat[4].s0, l_mat[5].s0, l_mat[6].s0, l_mat[7].s0);
22         src[size]   = (float8) (l_mat[0].s1, l_mat[1].s1, l_mat[2].s1, l_mat[3].s1,
       l_mat[4].s1, l_mat[5].s1, l_mat[6].s1, l_mat[7].s1);
23         src[2*size] = (float8) (l_mat[0].s2, l_mat[1].s2, l_mat[2].s2, l_mat[3].s2,
       l_mat[4].s2, l_mat[5].s2, l_mat[6].s2, l_mat[7].s2);
24         src[3*size] = (float8) (l_mat[0].s3, l_mat[1].s3, l_mat[2].s3, l_mat[3].s3,
       l_mat[4].s3, l_mat[5].s3, l_mat[6].s3, l_mat[7].s3);
25         src[4*size] = (float8) (l_mat[0].s4, l_mat[1].s4, l_mat[2].s4, l_mat[3].s4,
       l_mat[4].s4, l_mat[5].s4, l_mat[6].s4, l_mat[7].s4);
26         src[5*size] = (float8) (l_mat[0].s5, l_mat[1].s5, l_mat[2].s5, l_mat[3].s5,
       l_mat[4].s5, l_mat[5].s5, l_mat[6].s5, l_mat[7].s5);
27         src[6*size] = (float8) (l_mat[0].s6, l_mat[1].s6, l_mat[2].s6, l_mat[3].s6,
       l_mat[4].s6, l_mat[5].s6, l_mat[6].s6, l_mat[7].s6);
28         src[7*size] = (float8) (l_mat[0].s7, l_mat[1].s7, l_mat[2].s7, l_mat[3].s7,
       l_mat[4].s7, l_mat[5].s7, l_mat[6].s7, l_mat[7].s7);
29     } else {
30         dst = g_mat + col * size * 8 + row;
31
32         src[0]      = (float8) (dst[0].s0, dst[size].s0, dst[2*size].s0, dst[3*size
       ].s0, dst[4*size].s0, dst[5*size].s0, dst[6*size].s0, dst[7*size].s0);
33         src[size]   = (float8) (dst[0].s1, dst[size].s1, dst[2*size].s1, dst[3*size
       ].s1, dst[4*size].s1, dst[5*size].s1, dst[6*size].s1, dst[7*size].s1);
```

Listing B.9 Transposed OpenCL with AVX, transposition, continued

```
34          src[2*size] = (float8) (dst[0].s2, dst[size].s2, dst[2*size].s2, dst[3*size
        ].s2, dst[4*size].s2, dst[5*size].s2, dst[6*size].s2, dst[7*size].s2);
35          src[3*size] = (float8) (dst[0].s3, dst[size].s3, dst[2*size].s3, dst[3*size
        ].s3, dst[4*size].s3, dst[5*size].s3, dst[6*size].s3, dst[7*size].s3);
36          src[4*size] = (float8) (dst[0].s4, dst[size].s4, dst[2*size].s4, dst[3*size
        ].s4, dst[4*size].s4, dst[5*size].s4, dst[6*size].s4, dst[7*size].s4);
37          src[5*size] = (float8) (dst[0].s5, dst[size].s5, dst[2*size].s5, dst[3*size
        ].s5, dst[4*size].s5, dst[5*size].s5, dst[6*size].s5, dst[7*size].s5);
38          src[6*size] = (float8) (dst[0].s6, dst[size].s6, dst[2*size].s6, dst[3*size
        ].s6, dst[4*size].s6, dst[5*size].s6, dst[6*size].s6, dst[7*size].s6);
39          src[7*size] = (float8) (dst[0].s7, dst[size].s7, dst[2*size].s7, dst[3*size
        ].s7, dst[4*size].s7, dst[5*size].s7, dst[6*size].s7, dst[7*size].s7);
40
41          dst[0]      = (float8) (l_mat[0].s0, l_mat[1].s0, l_mat[2].s0, l_mat[3].s0,
         l_mat[4].s0, l_mat[5].s0, l_mat[6].s0, l_mat[7].s0);
42          dst[size]   = (float8) (l_mat[0].s1, l_mat[1].s1, l_mat[2].s1, l_mat[3].s1,
         l_mat[4].s1, l_mat[5].s1, l_mat[6].s1, l_mat[7].s1);
43          dst[2*size] = (float8) (l_mat[0].s2, l_mat[1].s2, l_mat[2].s2, l_mat[3].s2,
         l_mat[4].s2, l_mat[5].s2, l_mat[6].s2, l_mat[7].s2);
44          dst[3*size] = (float8) (l_mat[0].s3, l_mat[1].s3, l_mat[2].s3, l_mat[3].s3,
         l_mat[4].s3, l_mat[5].s3, l_mat[6].s3, l_mat[7].s3);
45          dst[4*size] = (float8) (l_mat[0].s4, l_mat[1].s4, l_mat[2].s4, l_mat[3].s4,
         l_mat[4].s4, l_mat[5].s4, l_mat[6].s4, l_mat[7].s4);
46          dst[5*size] = (float8) (l_mat[0].s5, l_mat[1].s5, l_mat[2].s5, l_mat[3].s5,
         l_mat[4].s5, l_mat[5].s5, l_mat[6].s5, l_mat[7].s5);
47          dst[6*size] = (float8) (l_mat[0].s6, l_mat[1].s6, l_mat[2].s6, l_mat[3].s6,
         l_mat[4].s6, l_mat[5].s6, l_mat[6].s6, l_mat[7].s6);
48          dst[7*size] = (float8) (l_mat[0].s7, l_mat[1].s7, l_mat[2].s7, l_mat[3].s7,
         l_mat[4].s7, l_mat[5].s7, l_mat[6].s7, l_mat[7].s7);
49      }
50  }
```

Listing B.10 Transposed OpenCL with AVX, multiplication

```
__kernel __attribute__((vec_type_hint(float8)))
void mult(__global float8 *a, __global float8 *b, __global float *c) {
    int a_row = get_global_id(0);
    int b_row = get_global_id(1);
    int num_rows = get_global_size(0);
    int vectors_per_row = num_rows / 8;

    a += a_row * vectors_per_row;
    b += b_row * vectors_per_row;

    float sum = 0.0f;
    for (int i = 0; i < vectors_per_row; i++) {
        sum += dot(a[i].lo, b[i].lo) + dot(a[i].hi, b[i].hi);
    }
    c[a_row * num_rows + b_row] = sum;
}
```

Appendix C

OmpSs kernels

This appendix contains the kernels in use in the applications presented in section 3.3.

Listing C.1 OmpSs kernel

```
1  #pragma omp task
2  void matmul_rbr_cbc_blocks(const float *a, const float *b, float *c, int row, int
       col, int n, int block_size, int number_of_blocks) {
3      int bb = block_size * block_size;
4      float *sums = calloc(bb, sizeof(*sums));
5  #if defined (SSE) || defined(AVX)
6      float dots[8];
7      int dot_mask = 0xff;
8  #endif
9      for (int block = 0; block < number_of_blocks; block++) {
10         int a_block_index = bb * (number_of_blocks * row + block);
11         int b_block_index = bb * (number_of_blocks * col + block);
12         for (int i = 0; i < block_size; i++) {
13             for (int j = 0; j < block_size; j++) {
14 #ifdef SSE
15                 for (int k = 0; k < block_size; k += 4) {
16 #endif
17 #ifdef AVX
18                 for (int k = 0; k < block_size; k += 8) {
19 #endif
20 #ifdef NONE
21                 for (int k = 0; k < block_size; k++) {
22 #endif
23                     int a_index = a_block_index + i * block_size + k;
24                     int b_index = b_block_index + j * block_size + k;
25 #ifdef SSE
26                     const __m128 _a = _mm_load_ps(&a[a_index]);
27                     const __m128 _b = _mm_load_ps(&b[b_index]);
28                     const __m128 _d = _mm_dp_ps(_a, _b, dot_mask);
29                     _mm_store_ps(dots, _d);
30                     sums[i*block_size+j] += dots[0];
31 #endif
32 #ifdef AVX
33                     const __m256 _a = _mm256_load_ps(&a[a_index]);
34                     const __m256 _b = _mm256_load_ps(&b[b_index]);
35                     const __m256 _d = _mm256_dp_ps(_a, _b, dot_mask);
36                     _mm256_store_ps(dots, _d);
37                     sums[i*block_size+j] += dots[0] + dots[4];
38 #endif
39 #ifdef NONE
40                     sums[i*block_size+j] += a[a_index] * b[b_index];
41 #endif
```

Listing C.2 OmpSs kernel, continued

```
42                 }
43             }
44         }
45     }
46     for (int i = 0; i < block_size; ++i) {
47         for (int j = 0; j < block_size; ++j) {
48             c[block_size * (row * n + col) + i*n + j] = sums[i*block_size + j];
49         }
50     }
51     free(sums);
52 }
```

Listing C.3 OmpSs kernel provided by BSC

```
1 #pragma omp task in([block_size][block_size] a, [block_size][block_size] b) inout([
      block_size][block_size] c)
2 void matmul_task(float *a, float *b, float *c, unsigned long block_size)
3 {
4     cblas_sgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans, block_size, block_size,
       block_size, 1.0, a, block_size, b, block_size, 1.0, c, block_size);
5 }
```

References

[1] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[2] Armin Ronacher, "Jinja2 (The Python Template Engine)." `http://jinja.pocoo.org`.

[3] "Top500 Supercomputer Sites." `http://www.top500.org`.

[4] A. Ramirez, "European scalable and power efficient HPC platform based on low-power embedded technology." `https://www.eesi-project.eu/media/BarcelonaConference/Day2/13-Mont-Blanc_Overview.pdf`, oct 2011.

[5] N. Rajovic, N. Puzovic, L. Vilanova, C. Villavieja, and A. Ramirez, "The low-power architecture approach towards exascale computing," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, pp. 1–2, ACM, 2011.

[6] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[7] "The Green500." `http://green500.org`.

[8] R. Giménez-Binder, "Mont-Blanc project selects Samsung Exynos 5 Processor." http://www.montblanc-project.eu/sites/default/files/sites/default/files/press-releases/press_release_montblanc_final.pdf, nov 2012.

[9] W. Kim, H. Chung, H. Cho, and Y. Kim, "Enjoy the Ultimate WQXGA Solution with Exynos 5 Dual (white paper)." http://www.samsung.com/global/business/semiconductor/minisite/Exynos/data/Enjoy_the_Ultimate_WQXGA_Solution_with_Exynos_5_Dual_WP.pdf, jul 2012.

[10] 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, IEEE, 2009.

[11] A. Munshi, "The OpenCL Specification." http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, nov 2012.

[12] "Khronos Adopters." http://www.khronos.org/conformance/adopters/adopter-companies.

[13] Nvidia, CUDA, "Programming guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2008.

[14] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, "Optimizing the exploitation of multicore processors and gpus with openmp and opencl," *Languages and Compilers for Parallel Computing*, pp. 215–229, 2011.

[15] 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, no. 02, pp. 173–193, 2011.

[16] "Programming Models @ BSC." http://pm.bsc.es.

[17] OpenMP Architecture Review Board, "OpenMP Application Program Interface."
`http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, jul 2011.

[18] M. Guillén Allés, "Openmp to opencl: Aprovechamiento de los recursos heterogéneos del sistema," Master's thesis, Universitat Politécnica de Catalunya, jun 2011.

[19] "UserManual/Schedule - NANOS++."
`https://pm.bsc.es/projects/nanox/wiki/UserManual/Schedule`.

[20] "Intel 64 and IA-32 Architectures Software Developer's Manual." `http://download.intel.com/products/processor/manual/325462.pdf`, mar 2013.

[21] "msr-tools."
`http://www.kernel.org/pub/linux/utils/cpu/msr-tools/`, jul 2004.

[22] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[23] ICL, University of Tennessee, "PAPI."
`http://icl.cs.utk.edu/papi/index.html`.

[24] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pp. 262–268, IEEE, 2012.

[25] S. Sharma, C.-H. Hsu, and W.-c. Feng, "Making a case for a green500 list," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 8–pp, IEEE, 2006.

[26] 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, IEEE, 1994.

[27] A. J. Martin, M. Nyström, and P. Penzes, "Et2: A metric for time and energy efficiency of computation," *Power-Aware Computing*, 2001.

[28] H. Lien, L. Natvig, A. Al Hasib, and J. Meyer, "Case studies of multicore energy efficiency in task based programs," *ICT as Key Technology against Global Warming*, pp. 44–54, 2012.

[29] ICL, University of Tennessee, "PAPI: The Low Level API." `http://icl.cs.utk.edu/papi/docs/dd/dbc/group__low__api.html`.

[30] Intel Corporation, "Using the Intel(R) SDK for OpenCL* - Offline Compiler Standalone Tool." `http://software.intel.com/sites/landingpage/opencl/user-guide/Using_the_Intel_SDK_for_OpenCL_Offline_Compiler_Standalone_Tool.htm`.

[31] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Performance tuning of matrix multiplication in opencl on different gpus and cpus," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 396–405, IEEE, 2012.

[32] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`http://www.netlib.org/lapack/lawns/lawn147.ps`).

[33] "Automatically Tuned Linear Algebra Software." `http://math-atlas.sourceforge.net`.