NTNU – Trondheim
Norwegian University of
Science and Technology

Energy Efficiency Experiments on Samsung Exynos 5 Heterogeneous Multicore using OmpSs Task Based Programming

Rune Holmgren

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Lasse Natvig, IDI
Co-supervisor: Antonio García Guirado, IDI

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

Abstract

This thesis explore the energy efficiency of task based programming with OpenMP SuperScalar (OmpSs) on the heterogeneous Samsung Exynos 5422 system on a chip. The system features small energy efficient cores, large high performance cores and a GPGPU, and OmpSs tasks were run on all three different processors. Experiments running a genetic algorithm and a Cholesky decomposition were used to gather results.

The option of running applications on the energy efficient cores, on the high performance cores or on a combination of processor cores and the GPU, allow programs to chose between energy efficiency or performance as necessary. The results showed that the energy efficient cores are the best option when performance is not an objective, while utilizing both high performance and energy efficient cores together gives the highest performance. For applications that are suited to utilize the GPU, it is possible to get a large increase in both performance and energy efficiency.

The complexity of developing parallel applications is one of the main obstacles of heterogeneous systems. The results showed that task based programming is a feasible solution to make development less complex, while maintaining both the parallel performance and energy efficiency of the system.

Problem statement

The CARD group has available several platforms with Exynos 5 MPSoCs (Multi-Processor System on Chip) from Samsung. These chips are used in numerous products, among them state-of-the-art high end smartphones and tablets. These heterogeneous multicores combine the ARM Cortex multicore (big.LITTLE) and the ARM Mali GPU developed in Trondheim.

Numerous research projects in Europe including the Mont Blanc project (www.montblanc-project.eu/) are using the OmpSs task based programming system. OmpSs is short for Open MP SuperScalar, and it extends OpenMP with an enhanced task concept to improve the portability of parallel programs (pm.bsc.es/ompss). The programming system is tailored both for CPUs and GPUs, and is an important contribution towards increased performance and programmability of heterogeneous multicores. OmpSs can dynamically handle several implementations of the same task, making it easy to explore the performance and energy efficiency of alternative parallelisation strategies, algorithms and implementations.

The project is focussed at doing performance evaluation experiments for getting insight in how parallelization to different cores can impact on performance and energy efficiency. The student should evaluate at least 2 different applications or benchmark-programs. It is a goal that both the CPU and GPU are used in at least one of the applications, and the GPU should be programmed using OpenCL. One of the applications can be a Genetic Algorithm computation. The other can be taken from the Mont Blanc application kernels, the Dimacs Implementation Challenges or some other well known benchmark suite.

The project is part of the EECS Strategic Research projects at IME (www.ntnu.edu/ime/eecs).

The project is reserved for student Rune Holmgren as a continuation of last years autumn project.

Main supervisor: Prof. Lasse Natvig

Co-supervisor: Toni Garcia, ARM - Trondheim

Acknowledgments

I would like to thank my primary supervisor Professor Lasse Natvig, and my co-supervisor Ph.D. Antonio Garcia. They have guided and supported me though all the challenges I have encountered in this final year of my degree. They have both enthusiastically followed my work and assisted me with both practical issues and supplied me with valuable feedback on my work. I am sincerely thankful for their help.

I would like to thank Trond Inge Lillesand. His master thesis on the usage of OmpSs and Neon/OpenCL for acceleration on ARM processors has been a valuable resource throughout my research.

Contents

Chapter 1

Introduction

In this chapter the motivation for the project, the project scope and goals, and thesis outline is presented.

1.1 Motivation

Hardware development today is facing many large issues that are holding back the technology. Single core is close to the wall for frequency, instruction-level parallelism and memory access. The increase in performance of cores have been slowing down, and development was forced to change focus. This led to the development of multicore processors. Unfortunately multicore systems are limited by the applications ability to run in parallel. This limits the number of cores that can be used before typical applications no longer gain performance. A solution to this problem might be heterogeneous systems with different specialized processor cores. The HiPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation) roadmap [1] consider the complexity of software development for heterogeneous systems one of the main challenges of transitioning from conventional shared memory multiprocessors to heterogeneous systems. OpenMP SuperScalar (OmpSs) is a proposed solution to solving this.

Energy efficiency is a problem that is currently limiting most areas in hardware technology. In development of new technology is limited both by the cost of power in terms of both battery life and higher power bills and systems producing too much heat. Achieving higher energy efficiency will enable lower power or higher performance in systems. Heterogeneous systems is an emerging technology that is a proposed solution to lowering power [2]. This thesis will explore the energy efficiency of the Samsung Exynos 5 heterogeneous multicore system on a chip (SoC). The SoC include four ARM Cortex-A7 low energy cores, four ARM Cortex-A15 high performance cores and an ARM Mali-T628 GPU. The performance and energy properties of all these devices will be explored.

1.2 Project Scope and Goal

The goal of this project is to explore the energy efficiency properties of task based programming on heterogeneous multiprocessors systems. Experiments with two test applications using OmpSs task based programming will be ran on the ODROID-XU3. The ODROID-XU3 features the heterogeneous Samsung Exynos 5422 SoC with two different ARM processor cores and an ARM GPU. The experiments will compare running on different configurations of the processors, including the GPU. The results should show the strengths and weaknesses of task based programming on heterogeneous systems with focus on the energy efficiency of the system.

1.3 Outline

This section gives a short outline of the report. Chapter 2 covers background. It covers the principles of energy efficiency, task based programming and OmpSs, heterogeneous systems and ARM big.LITTLE, the experiment platform and the test applications used in the experiments. Some sections of this chapter are from my own pilot project done the autumn of 2014 [3]. Chapter 3 present related work that can be relevant for the reader. Chapter 4 cover the setup and methodology used in the experiments. Information about how the experiments were run is covered in this section. Chapter 5 covers the implementation of the test applications used in the experiments. Chapter 6 present the results from the experiments and discuss them. Chapter 7 cover the conclusion that can be drawn from the results. Chapter 8 cover future work.

Chapter 2

Background

In this chapter covers background that is useful to understand the context of the problem statement, scope and goal of this thesis.

2.1 Energy efficiency

Energy efficiency is important for most computer systems today. Mobile devices needs to deliver performance with a long battery life, while server infrastructure need to deliver high performance with a low power bill.

Energy measurement

The interesting raw power properties of a system are voltage and current. In systems featuring sensors for such data, the data can be gathered while executing a program. By analyzing the data, it is possible to optimize the program for energy efficiency.

The power of the system is the product of the voltage and current, as shown in Equation 2.1. This is interesting to observe in applications that run when the system is standing by. When execution time does not matter, the energy consumed per second becomes the important metric.

$$Power = Current \times Voltage \tag{2.1}$$

Often the system is executing a program for a duration, and as soon as it is done the system can be shut down. In such cases, it is interesting to measure how much energy the system consumed during execution. The product of the system's power and the problem's execution time is as shown in Equation 2.2 the total amount of energy consumed.

$$Energy\ consumption(Joule) = Power(Watt) \times Execution\ time(Second) \tag{2.2}$$

There are many cases where a balance between energy efficiency and the performance trade off is interesting. This is when the energy delay product of the system becomes interesting. This is an interesting property for both scientific experiments and programs executed on regular systems. In research experiments, the goal may be to obtain energy efficiency or performance. In real life applications the goal will normally be a balance between the two. On regular systems in real life applications, both energy and performance matter. Users do not want to wait longer than necessary for the system to complete its task. Neither do they want the system to run out of battery or the power bill to grow too big.

$$Energy\ delay\ product = Energy \times Execution\ time \tag{2.3}$$

The energy delay product of a system is calculated by multiplying the execution time of the application with its energy, as shown in Equation 2.3.

2.2 Task based programming

Task based programming [4] allows a programmer to work with parallel programs, with abstraction from the parallelization itself. When programming with this model, the program can be split into tasks that can runs in parallel. When the program run, it will run with a runtime environment working as a task manager. This task manager can dynamically assign tasks to the processors, and the programmer does not have to handle all the time-consuming tasks related to manual parallelization. As long as the programmer correctly handles task dependencies in the parallelized code, it will be possible to write this kind of code as if it was serial.

The task based programming model also allows simpler development of portable programs. When the program is running tasks, it is not a problem to allow it to run on them on a larger or smaller numbers of processors. The processors can be conventional multi core processors, GPUs, heterogeneous multiprocessors or even clusters.

Several implementations of task based programming models exist, although most are prototypes used for research as of now. Examples are OmpSs which is central for this thesis and StarSs from which OmpSs takes many of it's features. There exist other implementations like ClusterSs which also include features from StarSs.

2.3 OpenMP Super scalar

OmpSs [5] is a task based programming model being developed at the Barcelona Supercomputing Center. It is an effort to extend OpenMP with features from the StarSs programming model, and support accelerator APIs like CUDA and OpenCL. OmpSs utilizes task level parallelism to support asynchronicity and heterogeneity on multiprocessors systems and cluster computers. The goal of OmpSs is to unify the development of applications for shared-memory processors, heterogeneous architectures and cluster computer systems. OmpSs features are used by OpenMP-like annotations in the code, which are used by the Mercurium source-to-source compiler to create calls to the runtime system. NANOS++ runtime system will receive these calls from the program and create tasks accordingly. The result is compiled to native code and linked with NANOS++ for an independent program.

OpenMP and StarSs

> "StarSs + OpenMP = OmpSs" [6].

OpenMP [7] is a portable parallel programming API for shared memory multiprocessors systems. It controls runtime behavior of programs through compiler directives, library routines and environment variables [8]. It is available on most processor architectures and operating systems, offering portability for parallel programs across these systems.

StarSs [9] is a programming model achieving task based programs through directives in sequential programs. It is extended to support a wide range of platforms including conventional shared-memory processors, GPGPUs, and clusters. OmpSs is an effort of extending OpenMP to support the features of StarSs in a single programming model.

The high expressiveness of OpenMP is maintained in the tasks of OmpSs while also using the automatic data transfers and runtime analysis of StarSs. OpenMP is built on a fork-join thread model, which is replaced by a scheduler maintained thread pool model in OmpSs. This makes OmpSs a task based programming model. In OpenMP, the programmer has to consider the underlying physical hardware available. In OmpSs this is replaced by declarations of dependencies, granting the advantages of task based programming discussed in Section 2.2

The Nanos++ runtime environment and the Mercurium compiler

NANOS++ [10] is a runtime environment for parallel programs running on shared memory systems. OmpSs programs run on top of Nanos++. Runtime parallelization of OpenMP programs is the main purpose of NANOS++, but it is designed to be flexible enough to support other programming models. One such programming model is OmpSs.

Mercurium is a source-to-source compiler used to translate OmpSs directives into Nanos++ task creation calls. After compiling OmpSs programs with Mercurium, they are compiled down to native code and linked with the Nanos++ Runtime Library by a conventional compiler. OmpSs directive calls will trigger Nanos++ library calls while running, and Nanos++ will execute them.

Scheduling

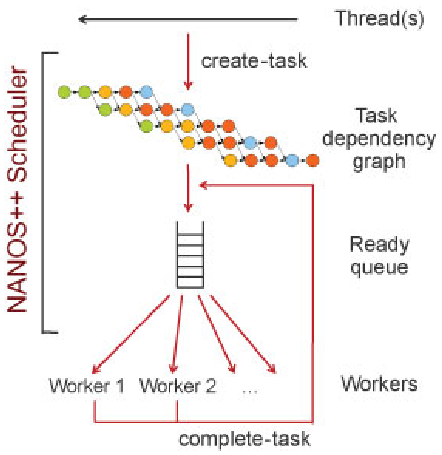


Figure 2.1: Nanos++ scheduler. Figure is taken from [11].

Figure 2.1 shows the workflow of the Nanos++ runtime environment used by OmpSs. The core in Nanos is a thread pool, a set of available workers and a dependency graph. The thread pool is managed by Nanos++ and is used to run tasks in parallel. OmpSs programs make calls to Nanos++ to create tasks. These tasks are added to the dependency graph. When a graph node has no parents in the dependency graph, it is available for workers in the ready queue. The workers are the available processing units. A free worker will be assigned an available task from the dependency graph. When the task is done, the worker will enter the ready queue and wait until it is assigned another task.

Using OmpSs

OmpSs programs are written mostly as serial programs, but with directives to trigger subroutines in OmpSs. The directives may also specify what device to target for the code. Heterogeneous multi-core processors or conventional SMPs are the most straightforward devices to use, but also accelerator APIs like CUDA and OpenCL can be targeted. These directives specify what parts of the code to run as tasks, what dependencies these parts have and on what device to run the code. It is also possible to add multiple implementations of a task targeting different devices. If an OpenCL implementation and an SMP implementation both exist, the scheduler will be able to use either device to run the task.

There are four dependencies that can be used OmpSs directives. An lvalue is the storage address where a variable is stored. It is used to unambiguous refer to the correct dependencies. Descriptions are from [5]:

input:

The task will not be eligible to run as long as a previously created task with an output clause applying to the same lvalue has not finished execution.

output:

The task will not be eligible to run as long as a previously created task with an input or output clause applying to the same lvalue has not finished execution.

inout:

The task that evaluates to a given lvalue is considered to have an input clause and an output clause that evaluates to the same lvalue.

inout-set:

Same as inout, except that it will not create dependencies to other tasks with and inout-set clause evaluating to the same lvalue.

By adding these annotations in OmpSs directives, the scheduler will be able to build a dependency graph like the one in Figure 2.1. The tasks are then executed in an order selected by the scheduler. This allows the programmer to focus on the problem, and let the scheduler avoid the challenges of race conditions by using the dependencies.

Listing 2.1: Example OmpSs task with inout dependencies and implementations for both SMP and OpenCL.

```
#pragma omp task inout([N]c_population, [N]n_population) no_copy_deps
void one_generation(int* c_population, int* n_population ...) {
  // C code for conventional SMP environment
}

#pragma omp target device(opencl) copy_deps implements(one_generation)
void one_generation_opencl(int* c_population, int* n_population ...) {
  // OpenCL kernel code
}
```

In Listing 2.1 an example of an OmpSs task with dependencies and multiple implementations is shown. When this function is called in an OmpSs program, the runtime environment will ensure that the dependencies are resolved and run it on an available device. If multiple calls are made in the program, the tasks will be able to run in parallel without further action by the programmer.

Scheduling

When a task is ready for execution, it is assigned to a worker by Nanos++. There are several scheduling policies that use different strategies to distribute these tasks. They differ in what tasks are given priority under which conditions, and what worker is assigned the tasks. The experiments of this thesis are run on a single-ISA heterogeneous processor, and because of this the bottom level-aware scheduler is used.

The following are the scheduling policies that are used for nonclusters (The descriptions are taken from [12]):

Breath first:

This scheduler policy only implements a single global ready queue. When creating a task with no dependences (or when a task becomes ready after all its dependences has been fulfilled) it is placed in this ready queue. Ready queue is ordered following a FIFO (First In First Out) algorithm by default, but it can be changed through parameters. Breadth first implements immediate successor mechanism by default (if not in conflict with priority).

Distributed breath first:

Is a breadth first algorithm implemented with thread local queues instead of having a single/global ready queue. Each thread inserts its created tasks into its own ready queue following a FIFO policy (First In First Out, inserting in queue's front and retrieving from queue's back) algorithm. If thread local ready queue is empty, it tries to execute current task's parent (if it is queued in any other thread ready queue). In the case parent task cannot be eligible for execution it steals from next thread ready queue. Stealing retrieves tasks from queue's front (i.e. the opposite side from local retrieve).

Work first:

This scheduler policy implements a local ready queue per thread. Once a task is created it chooses to continue with the new created task, leaving current task (creator) into current thread's ready queue. Default behaviour is implemented through FIFO access to local queue, LIFO access on steal and steals parent if available which actually is equivalent with the cilk scheduler behaviour.

Socket-aware scheduler:

This scheduler will assign top level tasks (depth 1) to a NUMA node set by the user before task creation while nested tasks will run in the same node as their parent. To do that, the user must call the nanos_current_socket function before executing tasks to set the NUMA node the task will be assigned to. The queues are sorted by priority, and there are as many queues as NUMA nodes specified (see num-sockets parameter). Besides that, changing the binding start and stride is not supported. Work stealing is optional. By default, work stealing of child tasks is enabled. Upon initialisation, the policy will create lists of valid nodes to steal. For each node, the policy will only steal from the closest nodes. Use numactl –hardware to print the distance matrix that the policy will use. For each node, a pointer to the next node to steal is kept; if a node steals a task, it will not affect where other nodes will steal. There is an option to steal from random nodes as well, and to steal top level tasks instead of child tasks.

Bottom level-aware scheduler:

This scheduler targets single-ISA heterogeneous machines that maintain two kinds of cores (fast and slow, such as ARM big.LITTLE). The scheduler detects dynamically the longest path of the task dependency graph and assigns the tasks that belong to this path (critical tasks) to the fast cores of the system. The detection of the longest path is based on the computation and the usage of bottom-level longest-path priorities, that is the length of the longest path in the dependency chains from each node to a leaf node. There are two queues for the ready tasks, one per processor kind. Fast cores retrieve tasks from their unique queue and slow cores retrieve tasks from the other queue. The queues are sorted according to the task priority (bottom level). Work stealing is enabled by default for fast cores: a fast core steals a task from the slow-cores' queue if it is idle. Optionally, work stealing can be performed by both sides if the parameter NX_STEALB is enabled. The policy can be flexible or strict, meaning that the flexible policy considers more tasks as critical, while the strict limits the number of critical tasks.

2.4 Heterogeneous systems

Conventional multicore shared memory processors have several identical cores. Accelerators like GPUs are connected to an external bus and have no direct communication to the processor cores or their memory hierarchy. It is irrelevant which of the processors are used to run a process or thread. Adding more processors increase parallel processing power linearly. Heterogeneous systems have dissimilar processors working together. The different cores typically incorporate specialized features to handle particular tasks. Different tasks can be assigned to suitable processors to increase their performance or energy efficiency. Multiple or all of the processors in heterogeneous systems can be configured in a shared-memory architecture. They will then have access to data from each other without spending time transferring the data on a bus.

An emerging use of heterogeneity is single ISA-architectures [2]. Single ISA heterogeneous architectures have multiple dissimilar processors with a common ISA. A programs is able to run on any of the processors without altering the program. They typically use cores supporting all of the same features, but with different performance and energy efficiency properties. Single ISA heterogeneous architectures can increase energy efficiency by 39% percent while sacrificing 3% performance compared to traditional homogeneous architectures. By optimizing the Scheduling for energy delay product, an energy delay product can be improved by a factor of 3 while sacrificing 22% performance [2].

Another promising use of heterogeneity is heterogeneous-ISA architectures. Heterogeneous-ISA architectures have dissimilar cores with different ISAs. A program for one ISA can not run on the processors with different ISAs unless it's changed. A common use of heterogeneous-ISA architecture is to add a GPU among the processor cores. The GPU require programs written in another way than conventional programs, but the programs are able to utilize a high level of parallelism. This GPU is available on the same bus as the other cores, and they share a common bus and memory system. Figure 2.2 shows an example of such an interconnected bus. Assigning tasks to the GPU can be very efficient on such systems because there is less overhead from data transferring over the bus than in conventional systems. Heterogeneous-ISA architectures can increase performance by 21%, energy savings by 23% and reduce the energy delay product by 32% compared to single ISA architectures [13].

The Samsung Exynos 5422 used in this thesis features both ARM Cortex-A15 and ARM Cortex-A7 cores that share a common ISA, as well as an ARM Mali-T628 GPU. All these cores are interconnected in a shared-memory system with a common connection.

2.5 ARM big.LITTLE

ARM big.LITTLE [14] is a power-optimization technology that allow ARM processors with a common instruction set and feature set, but with different performance and energy efficiency, to cooperate. A system with ARM big.LITTLE will feature high-performance processor cores, which meet the performance demand from modern systems. It will also feature energy efficiency tuned low power cores with lower performance, which meet the energy efficiency demand of modern systems. It is possible to run high performance applications on the large processors, or even both the large and small processors simultaneously. Applications demanding less performance can be run without use of the high-performance cores. In this way the system can save power, and increase battery life or lower the power bill of the system.

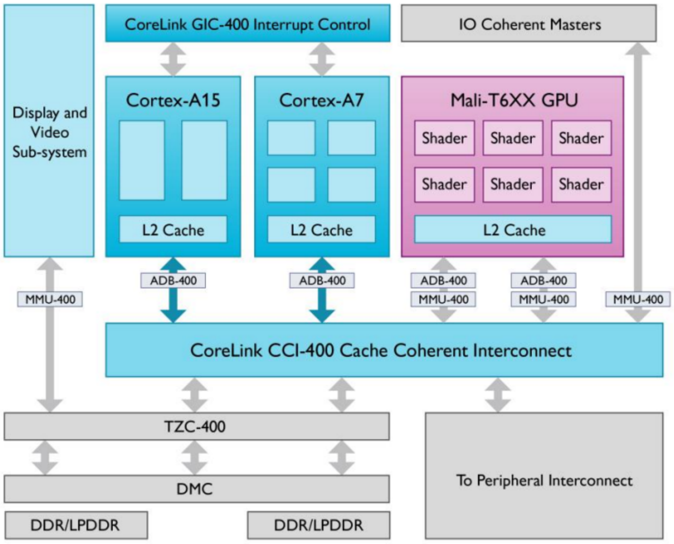


Figure 2.2: Big.LITTLE architecture with cache coherent interconnected ARM Cortex-A15 cores, ARM Cortex-A7 cores and an ARM Mali-T6XX GPU. The figure is from [15].

When utilizing systems with ARM big.LITTLE, it is not necessary for the programmer to specifically program for this system. Big.LITTLE MP is an underlying software that will automatically and seamlessly move workloads between processor cores. When the system is running general parallel problems, the added small cores in the system can give a performance increase of over 40% [16]. At the same time, tasks with low performance demand can exhibit power savings in the range of 30%-50% [16] on the SoC level. In Figure 2.2 the architecture of a typical ARM big.LITTLE system with Cortex-A15 cores and Cortex-A7 cores is shown.

2.6 Experiment platform

In this section the properties and features of the ODROID-XU3 experiment platform will be elaborated.

2.6.1 ODROID-XU3

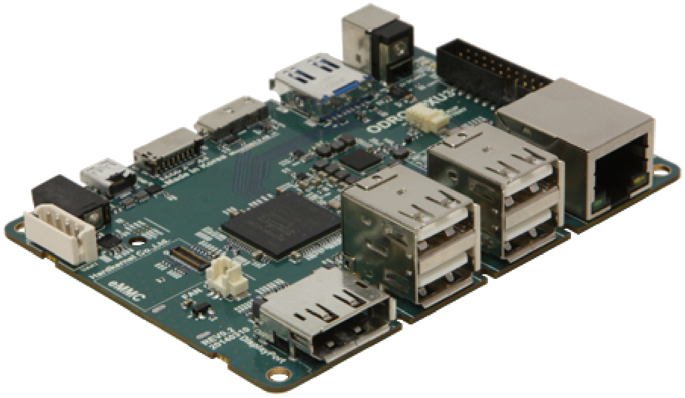


Figure 2.3: ODROID-XU3

The ODROID-XU3 is a new single-board computing system depicted in Figure 2.3, offering interesting properties for these experiments. The system has an Exynos 5422 heterogeneous SoC. Exynos 5422 has a quad core ARM Cortex-A15 CPU and an ARM Mali-T628 GPU, but also a smaller quad core ARM Cortex-A7 processor. These 3 different processing units can be used simultaneously to solve problems. They use ARMs big.LITTLE architecture, described in section 2.5, to achieve this heterogeneous cooperation. In this thesis the performance and energy efficiency of this kind of heterogeneous processor will be explored. This is the system that will run most of the experiments, including all energy efficiency experiments. In Table 2.1, the system specifications of the board are listed.

SoC	Samsung Exynos 5422
CPU 1	
Model	ARM Cortex-A15
Manufacturing process	32nm
Maximum clock frequency	2.0GHz
Number of cores	4
L2 Cache	512KB
L1 Cache	32KB/32KB I/D
CPU 2	
Model	ARM Cortex-A7
Manufacturing process	32nm
Maximum clock frequency	1.4GHz
Number of cores	4
L2 Cache	2MB
L1 Cache	32KB/32KB I/D
GPU	
Model	ARM Mali-T628 MP6
Maximum clock frequency	600 MHz
Memory	
Available memory	2 GB
Maximum clock frequency	933MHz
Operating system	
Distribution	Linux odroid
Version	3.10.54+

Table 2.1: ODROID-XU3 specifications

Performance

The ODROID-XU3 features a Samsung Exynos 5422 system on a chip. There are four ARM Cortex-A15 running at up to 2GHz and four ARM Cortex-A7 running up to 1.4GHz. These cores offer both high performance through the ARM Cortex-A15 cores and low energy computations through the ARM Cortex-A7. The 2GB of RAM is LPDDR3 RAM running at 933MHz yielding 14.9GB/s memory bandwidth[17]. The ARM Mali-T628 is a versatile GPU with support for many popular APIs for graphics and GPU computation. It is clocked to 600MHz in the Samsung Exynos 5422.

More details on the CPU cores can be found in Sections 2.6.2 and 2.6.3, and more information about the GPU can be found in Section 2.6.4

The ODROID-XU3 features the new eMMC 5.0 standard for storage. The performance of this standard outperforms both older eMMC standards, as well as memory card readers, which other similar systems may contain. The ODROID-XU3 can achieve a read/write performance of 198/74 MB/s[17], which a lot better than older systems like the Arndale Board.

2.6.2 ARM Cortex-A15

The ARM Cortex-A15 is a high performance processor used in a wide range of mobile and embedded devices. The specifications of the core are listed in Table 2.2. Systems can implement these cores with many other devices. It support many performance and energy efficiency boosting technologies, including DSPs, GPUs and ARM big.LITTLE.

Frequency	1.0 GHz to 2.5GHz
L1 Cache	64KB
L2 Cache	4 MB
L3 Cache	None in core, may be implemented shared in multi core system.
Architecture	ARMv7-A
Architecture	ARMv7-A
Supported features	TrustZone® security technology NEON™ Advanced SIMD DSP & SIMD extensions VFPv4 Floating point Hardware vitalization support Integer Divide Fused MAC Hypervisor debug instructions
Memory management	40-bit ARMv7 Memory Management Unit

Table 2.2: ARM Cortex-A15 specifications

2.6.3 ARM Cortex-A7

The ARM Cortex-A7 is designed to be a low power alternative to the ARM Cortex-A15 and ARM Cortex-A17, with the same supported ISA and features. The specifications of the core are listed in Table 2.3. This enables the ARM Cortex-A7 to be paired with its larger relatives in an ARM big.LITTLE configuration. The Exynos 5422 SoC on the ODROID-XU3 board features 4 of these cores.

Frequency	1.2 GHz to 1.6GHz
L1 Cache L2 Cache L3 Cache	8-64KB up to 1 MB None in core, may be implemented shared in multi core system.
Architecture	ARMv7-A
Supported features	ARM Thumb-2 TrustZone® security technology NEON™ Advanced SIMD DSP & SIMD extensions VFPv4 Floating point Hardware vitalization support Integer Divide Fused MAC Hypervisor debug instructions
Memory management	40-bit ARMv7 Memory Management Unit

Table 2.3: ARM Cortex-A7 specifications

2.6.4 ARM Mali-T628

This is a high performance low power mobile GPU. The specifications of the GPU are listed in Table 2.4. The ODROID-XU3 board features one of these in it’s Exynos 5244 SoC.

Frequency	533/695 MHz 17/23.7 GFLOPS
Multi core support	1-8 cores
API Support	OpenGL 1.1, 2.0, 3.0 and 3.1 OpenCL 1.1 DirectX 11 RenderScript
Anti-Aliasing	4xFSAA with minimal performance drop 16xFSAA
Cache	32-256KB L2 cache

Table 2.4: ARM Mali-T628 specifications

2.7 Test applications

In this section the background of the two test applications will be introduced.

2.7.1 Genetic algorithm

Genetic algorithms are search heuristics inspired by evolution. They have applications within optimization and search problems. Problems where genetic algorithms are suited typically have a large solution space with candidates consisting of a set of parameters. Candidate solutions must be possible to encode as a string of genes similar to a chromosome. With a solution space of such chromosomes, a genetic algorithm will be able to explore and exploit solution candidates. A genetic algorithm works on a population of individuals, each containing a chromosome. It will mutate individuals and recombine the population of solution candidates while the evolutionary behavior will preserve information in the genes that are effective for solving the problem. By repeating this process and selecting parents with a bias to strong candidates, the individuals will normally develop into better and better solutions to the problem.

There are many variations of genetic algorithms that improve or alter them in different ways. Most of these variations are outside the scope of this thesis. The canonical genetic algorithm is what will be explored, as well as the island model for parallelization of the genetic algorithm. Figure 2.4 shows the canonical genetic algorithm. It is the typical textbook variation of a genetic algorithm introduced by John Holland and his students in 1975[18].

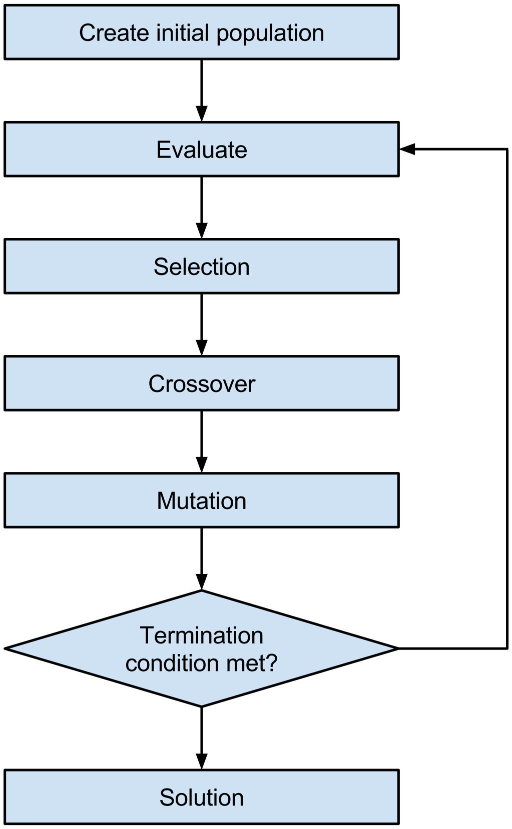


Figure 2.4: The steps of the canonical genetic algorithm.

All genetic algorithms need an initial population to work on. The individuals for this initial population are typically generated randomly [19]. The individuals in the canonical genetic algorithm are bit strings of a specific length. Each bit represents a parameter of the problem and the evaluation function. Not all problems have solution candidates that are easily represented as a bit string. Often a string of numbers or symbols is used. In genetic algorithms where the problem demands solutions on a specific form the individuals may be generated by an algorithm. An example of this is the traveling salesman problem, where you have to make sure that each town is visited once and only once [20]. Finding a suited encoding of the solution candidates of a problem, is normally one of the two problem dependent challenges of using genetic algorithms to solve it. The other is creating an evaluation function.

The first step of the main loop in a genetic algorithm is to evaluate all the individuals. When a genetic algorithm is used to search for a solution to a problem, there must be some way to determine how good a solution is. The fitness function for a problem will take a single individual as input and generate a score representing how good individual is as a solution candidate. Comparing the fitness function of two individuals should reflect how good they are compared to each other. Creating an evaluation function that accurately represents how good a solution candidate is, can be challenging. An evaluation function that does not reflect well enough upon good solutions, risk missing promising areas of the solution space.

When the population has been evaluated, the next three steps in the genetic algorithm will generate the next generation of individuals. First is the selection step. The next generation needs to inherit properties from the last generation to be able to improve over time. Selecting random individuals with a bias for selecting strong individuals is the normal approach. Many strategies for selecting these individuals exist. In this thesis, a wheel of fortune strategy was used. A range from 0 to the sum of the population's fitness values is divided such that each individual gets a portion the size of it's own fitness value. Then a random point in the range is selected, and the individual that owns that portion of the range gets selected. This is repeated until enough individuals for the next generation have been selected. Strong individuals are likely to get selected multiple times while weak individuals are likely to get selected only rarely.

The next step in generating the next generation is the crossover step. To gain the strengths of the previous generation, two parents from the current generation are used in generating two children for the next generation. Their genes are combined, picking some bits from one parent and some from the other. Each gene in their gene string is given a random chance of being swapped between the two individuals. After all genes are swapped or left intact at random, two new individuals have been generated. These new individuals are then added to the next generation.

The final step in generating the next generation is the mutation step. In this step, small changes are made to random individuals to make sure that the genetic algorithm is able to discover new areas of the solution space. At a set mutation rate, each gene in each individual, is given a chance to change randomly. The mutation rate is typically small enough to only make minor changes to some individuals, to make sure that not all good solution candidates are ruined. It still needs to be large enough to make sure that new areas of the solution space is explored.

When the next generation is done, it is time to check if the termination condition of the genetic algorithm is met. If they are not met, another generation of the genetic algorithm will be started. To stop the algorithm from running too long, a maximum number of generations may also be used. The algorithm will then terminate when the maximum number of generations has been reached, and the gene string of the best individual that has been found will be used as the solution. It is normal to have some target fitness value for the genetic algorithm to reach before termination. When an individual with a higher fitness value than the target fitness value has been found, that individual's gene string is the solution. The target fitness value is typically high enough to be used as a solution to the problem. This is because the perfect fitness value for problems solved by genetic algorithms is often unknown, and because the problems are too hard to be solved optimally. In some other cases only a perfect solution is accepted.

Island model

The island model for genetic algorithms will split the population into several subpopulations. It occasionally allow individuals to migrate between the subpopulations. By doing these exchanges, development can spread between the islands. Several processors can work on different subpopulations at the same time, and only occasionally synchronize with each other. The result is a low degree of overhead from the parallelization at the same time as the efficiency of the genetic algorithm is maintained.

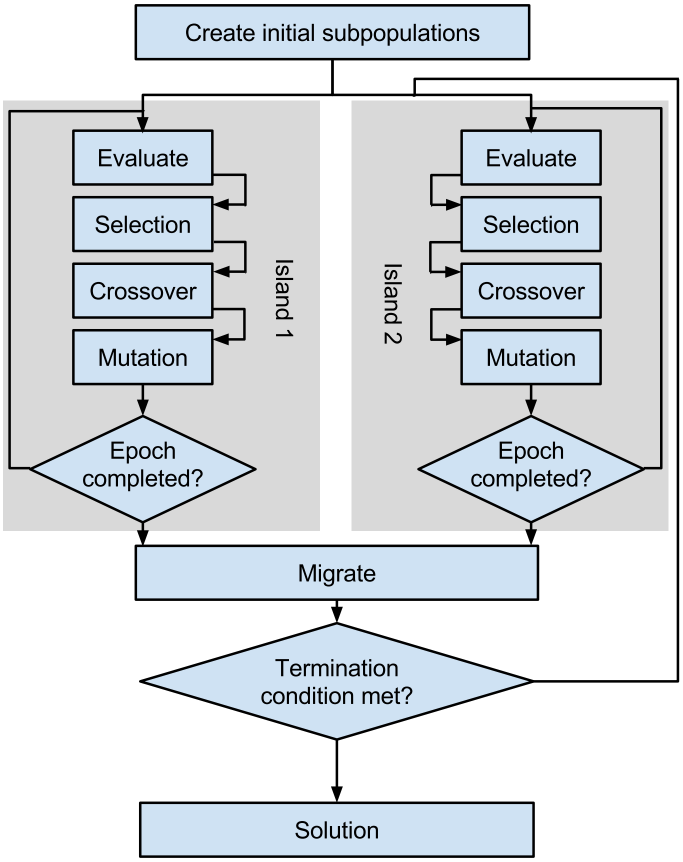


Figure 2.5: The island genetic algorithm shown for two subpopulations or islands.

Figure 2.5 shows the sequence of steps in the genetic algorithm with the island model. As all other genetic algorithms, it starts by generating a population of individuals. This population is divided into several subpopulations. For problems requiring a large amount of work in the generation of the individuals, it is possible to parallelize the generation of the population on multiple processors. When the population is done, the main part of the program is started. This is the part that runs in parallel. An island will run the genetic algorithm for each subpopulation. The genetic algorithm on each island will run independent from the other islands for a number of generations called an epoch. The evaluation, selection, crossover and mutation steps running on each subpopulation are identical to the ones described in Section 2.7.1. When the epoch is over, the islands need to exchange individuals. There are several different schemes to determine what individuals to move where. A common scheme is to select individuals at random with a bias for individuals with a high fitness value. These individuals will then be moved to some other island. The pattern they are moved in between the islands may be a complete graph, a grid in some number of dimensions, a ring or some other pattern depending on what is best suited for the problem and platform. When the migration is complete, the termination conditions are checked. If they are not met, another epoch is started on the islands. This is repeated until the conditions are met, and the program terminates.

2.7.2 Cholesky decomposition

Cholesky decomposition, also called Cholesky factorization, is a matrix decomposition used in linear algebra. It is commonly used to solve systems of linear equations on the form Ax = b, where A has to be a symmetric positive-definite matrix. When this requirement is not met, there are other decompositions available. These decompositions are either more complex to calculate or they have other requirements for the input matrix. Other applications of Cholesky decomposition include non-linear optimization, Monte Carlo simulations and Kalman filters.

$$A = LL* \tag{2.4}$$

Equation 2.4 shows the form of the Cholesky decomposition. A is the positive-definite input matrix representing the problem. The matrix may contain both real and complex numbers. If there are complex numbers present, the matrix must be a Hermitian positive-definite matrix. L is the lower triangular matrix and L* is the conjugate transpose of L. For any A with only real numbers L will only contain real numbers. This means that L* only needs to be the transpose matrix of L, and the form will change as shown in :

$$A = LL^{\intercal} \tag{2.5}$$

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & & 0 \\ \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \dots & l_{n1} \\ 0 & l_{22} & & l_{n2} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & l_{nn} \end{bmatrix} \tag{2.6}$$

Equation 2.6 shows the individual items in the matrices from Equation 2.5. Cholesky decomposition solves this equation by calculating each individual item in L and L*. Equation 2.7 and Equation 2.8 show formulas for calculating the value of each item.

$$\text{For k from 1 to n} \quad l_{kk} = \sqrt{a_{kk} + \sum_{s=1}^{k-1} l_{ks}^2} \tag{2.7}$$

$$\text{For j from k+1 to n} \quad l_{jk} = \frac{a_{jk} - \sum_{s=1}^{k-1} l_{js} l_{ks}}{l_{kk}} \tag{2.8}$$

Performance

The Cholesky decomposition algorithm is able to calculate the lower triangular matrix with a complexity of $n^3/3$, where n is the size of matrix A. This is twice as fast as the LU decomposition algorithm which use $2n^3/3$ [21]. This complexity is useful because it enable us to calculate the FLOPS of an application. In this thesis the FLOPS measurements will be used to evaluate the performance of the application.

Chapter 3

Related work

This section highlights existing work related to subjects discussed in this thesis.

3.1 Heterogeneous multicore systems

As we are approaching the limits of performance and energy efficiency of multiprocessor systems. Amar Shan [22] addresses heterogeneous processors as a proposed solution. By utilizing multiple heterogeneous cores, it is possible to overcome the limits of traditional multiprocessor systems. A heterogeneous system can include GPGPUs along side their conventional processor cores. Ryoo et al. [23] addressed the potential of general purpose computing on graphics processing unit. They showed that GPGPUs deliver tremendous computational power and memory bandwidth.

3.2 OpenMP Super Scalar (OmpSs)

Duran et al. [5] proposed OmpSs as an implementation of the task based programming model [4]. It extends OpenMP with new directives to support running parallel tasks. The objective of OmpSs is to extend OpenMP to support heterogeneous and asynchronous parallelism. It also enables parallel programs to utilize accelerators like OpenCL in the model.

Lien et al. [24] performed a case study on the performance and energy efficiency of parallelization with task based programming and vectorization. The experiments were run on both Intel Sandy Bridge and ARM Cortex-A9 processors. Their work shows that both parallelization through task based programming and vectorization can increase performance and energy efficiency and that the nature of the problem and scheduling will affect the results.

3.3 Genetic algorithms

Genetic algorithms were chosen as a test application because for the combination of parallelizable steps and steps requiring synchronization. The applications of genetic algorithms are addressed by Tang et al. [25]. Exploration of the energy efficiency of genetic algorithms is interesting because it also have applications in mobile devices. Sedaaghi et al. [26] explored the usage of genetic algorithms for speech recognition. Whitley et al. discusses the use of several subpopulations isolated on islands as an approach to parallelization of genetic algorithms. The technique showed good results and has become the standard for parallelization of genetic algorithms.

3.4 Cholesky decomposition

George [27] addresses the parallelization of Cholesky decomposition on shared memory multiprocessor systems. Several approaches are addressed, and the promising column-Cholesky algorithm has since become the normal approach for parallel Cholesky decomposition.

Chapter 4

Setup and Methodology

This chapter gives an overview of the software and methods used to collect data. The main purpose of this is to be able to reproduce the experiments for further research in the area.

4.1 Software

This is the software used in for compiling and running the experiments used in this thesis. It is included for the purpose of being able to reproduce the results.

Software	Version
gcc	4.8.2
mcc & oclmmc	1.99.4
Nanos++	0.9a
Mali OpenCL SDK	1.1.0

Table 4.1: ODROID-XU3 specifications

4.2 Compilation and running of applications

These are the commands used to compile the test applications used in this thesis. This section contains all the compilers, flags and environment variables needed to compile and run the genetic algorithm used in this thesis. The Cholesky decomposition is supplied with a Makefile for compilation, and is not included.

Before running the OmpSs test applications, the arguments for the Nanos++ runtime environment should be set. These are the values used in the execution of the test applications in this thesis. The summary flag will make Nanos++ provide useful information about the execution, it does not affect the execution itself and can be removed. The schedule flag will choose the

Serial genetic algorithm:

```
gcc simple_ga.c -std=c99
```

CPU only OmpSs genetic algorithm:

```
mcc island_ga.c -lgsl -lgslcblas -ompss -mfpu=neon
```

Manual OpenCL genetic algorithm:

```
mcc -c -Wall -I/home/rune/Mali_OpenCL_SDK_v1.1.0/include
-I/home/rune/Mali_OpenCL_SDK_v1.1.0/common -I. -lgsl
-lgslcblas -ompss -mfpu=neon ga.c -o ga.o
mcc ga.o -o ga -L/home/rune/Mali_OpenCL_SDK_v1.1.0/lib
-L/home/rune/Mali_OpenCL_SDK_v1.1.0/common -lOpenCL -lCommon
-lgsl -lgslcblas -ompss -mfpu=neon
```

CPU and OpenCL OmpSs genetic algorithm:

```
oclmcc ga.c ga.cl -lgsl -lgslcblas -ompss -mfpu=neon -std=c99
```

Table 4.2: Test

scheduler used by Nanos++. The Bottom Level Scheduler was used for all of the experiments in this thesis.

```
NX_ARGS='-summary -schedule=botlev'
```

The executables can be run with no flags.

4.3 Configuring processor cores

The processor cores of the Exynos 5422 can be turned on and off with the system running. Turning off a processor core is done by the operating system without the process running in that processor noticing that it was moved. Because of this it is possible to run different parts of a program in different processor configurations.

Turning on and off a processor core is done by writing wither a 0 or 1 to /sys/devices/system/cpu/cpuN/online. Processor 0-3 are the 4 ARM Cortex-A7 cores. Processor 4-7 are the 4 ARM Cortex-A15 cores. It is not possible to turn processor 0 off by writing a 0 to the corresponding file. Because of this it is left running in all experiments. In future work on the platform, other methods for turning off processor 0 may be tried. This is an example of a command that will turn off processor 5:

```
sudo bash -c 'echo 0 > /sys/devices/system/cpu/cpu5/online'
```

4.4 Performance measurement

In the experiments used for this thesis, execution time was used as the primary metric for performance. While running the experiments, the POSIX function gettimeofday() is used before and after running the region of interest. The time difference is a used as a measurement of how good the performance is.

4.5 Energy measurement on ODROID-XU3

As elaborated in Section 2.6.1, the ODROID-XU3 features four current sensors. These current sensors are used to monitor the current flow to the four large large cores, the four small cores, the GPU and the memory respectively. In addition to the current sensors, it is possible to read the supply voltage to each of these components. Based on these sensor readings, we can calculate the power consumption of each of the components in real time while running our applications, as shown in Equation 4.1. Using this scheme for power measurement, we can get readings with high precision of each component. The results will show how the consumption varies with different application configurations both for the application as a whole and for different stages of execution.

$$Power(Watt) = Current(Ampere) \times Voltage(Volt) \tag{4.1}$$

There is no way to avoid affecting performance with the energy measurements. To keep accuracy high with minimal interference with the problem execution, a delay of 200ms between each measurement was used. This was because test runs of the experiments showed that shorter delays would start to affect the performance too much. The reason for the energy measurements affecting the processor is that the program collecting the data runs on the CPU.

Chapter 5

Implementation

This section describes the Implementation of the genetic algorithm and Cholesky test applications, using OmpSs and OpenCL.

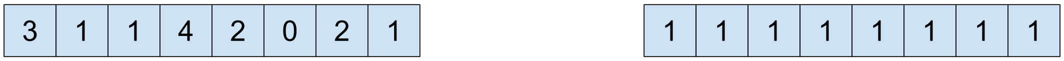
5.1 Genetic algorithm

A genetic algorithm is one of the two test applications for this thesis. It was chosen because it is a problem that includes both steps requiring synchronization and asynchronous steps that are highly parallelizable. This makes the problem interesting because the size of the parallel part may affect the performance gain or loss by utilizing different parallelization techniques.

Two different versions of this test application were implemented from scratch. A canonical genetic algorithm implementation made by John LeFlohic for use in education at Stanford University was used as a reference for the implementations [28]. The first is a CPU only OmpSs implementation which runs on between one and all of the available processor cores. The second is an CPU and OpenCL OmpSs implementation which can run the fitness evaluation for the whole population as a highly parallel OpenCL kernel, while the generation of the next generation of individuals is done in parallel on the available CPUs.

5.1.1 Problem and solution encoding

Any genetic algorithm needs to have a problem to solves. In this thesis, the interesting aspect of the genetic algorithm is not the problem it solve. A simple problem called one maximum was used. In one max, the goal is to create a perfect individual containing only 1s. The perfect individual is shown in Figure 5.1b.



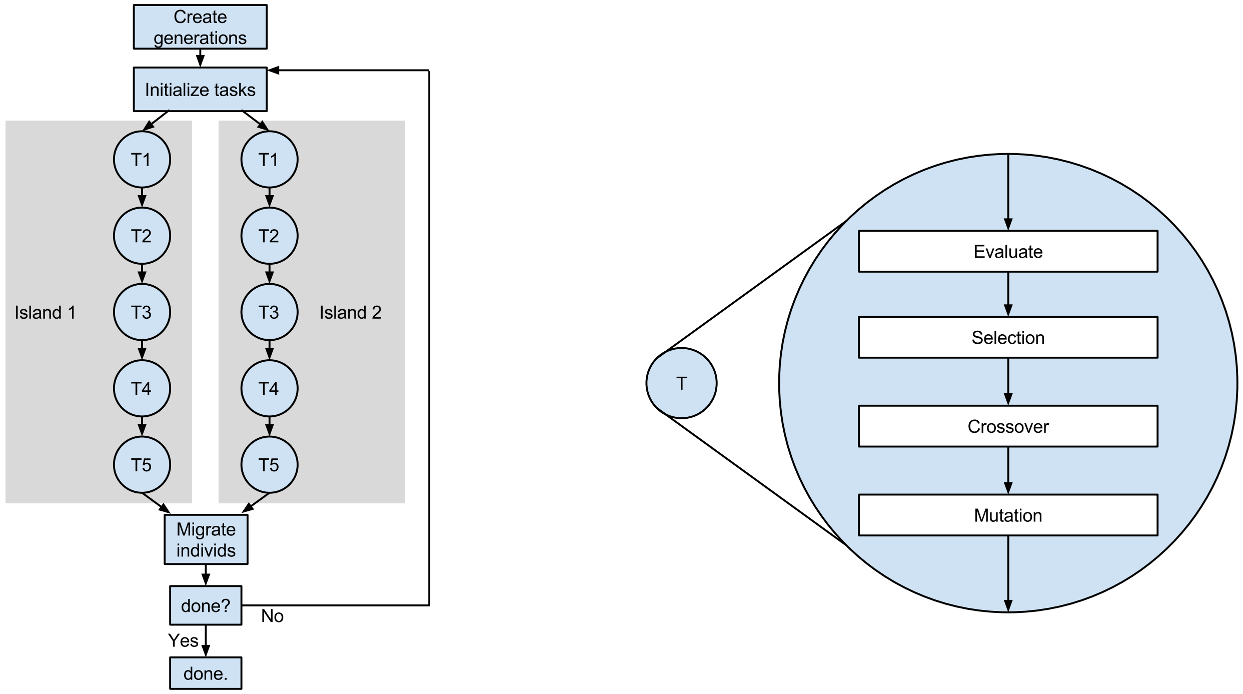
(a) A random individual.

(b) The perfect individual.

Figure 5.1: A random and the perfect individual from the one max problem used in the genetic algorithm test application. In this example, each gene is a number between 0 and 3, and the chromosomes are eight genes long.

5.1.2 OmpSs implementation using the CPUs only

As discussed in Section 2.7.1 the island model can achieve good results even when running on a single processor. It also offers a way to parallelize genetic algorithms with little overhead. Because of these advantages the model was chosen for the OmpSs task based implementation.



(a) The steps from the genetic algorithm that is run in each task. One task is one generation of the genetic algorithm executed on the islands subpopulation.

(b) The program flow of tasks shown in the genetic algorithm in Figure 5.2a.

Figure 5.2: The program flow of the OmpSs genetic algorithm test application. This version uses only the CPUs in parallel using the island model. Two islands and an epoch length of 5 generations are shown in the figure. In the test application these parameters are varied as needed.

Figure 5.2 shows the program flow of the OmpSs genetic algorithm implementation using only the CPUs. It implements the algorithm that is shown in Section 2.7.1 with OmpSs tasks. Different numbers of islands were tested, and the results showed that there should be two islands for each available processor core. This is to allow there to be a pool of tasks at any time for the workers to execute. This way the big cores can't complete their own island long before the small cores and waste processor time waiting necessary. The results of these tests are presented and discussed in greater detail in Section 6.1.2. The program will first create a random population. It will then initialize the OmpSs tasks for running one epoch of the genetic algorithm, shown as T1 to T5 in Figure 5.2a. Each island will have a chain of OmpSs tasks running with it's subpopulation as input and output. Each task is a generation of the genetic algorithm, running all the steps shown in Figure 5.2b. OmpSs will then distribute these tasks to the available processor cores. After all tasks have been completed, there will be a phase where individuals migrate between the subpopulations. In the end, we check the best fitness and compare it to the target fitness to determine if the program should terminate. Until this check succeed, or the max number of generations is reached, the program will continue to run series of generations.

The genetic algorithm will operate with sub populations contained on islands. In the figure, two islands are shown, this is a number set on the start of the program and will be equal to the number of processors utilized. The chain of tasks on each island is independent of all the other chains. This allows the parallel execution on each processor core to be completely independent of each other with no synchronization. The performance of the program can scale well on parallel machines.

Between each synchronization, migration and check of the algorithms completion conditions, each subpopulation will run several generations. The number of generations between each synchronization is a parameter that can be varied. In Figure 5.2 the number is set to 5. There are two reasons for executing several generations in each iteration. In this thesis, heterogeneous processors are central. OmpSs can evaluate what processor cores to use for different tasks. If there was only one task running on each island the high performance ARM Cortex-A15 cores would complete their task earlier than the slower ARM Cortex-A7 cores, and performance and energy would be wasted waiting for the slower cores to complete their tasks. By running several tasks, each executing a single generation, OmpSs can assign more generations to the high performance cores. The other reason is the overhead of running in parallel. There is an overhead each time the processors have to wait for all other processors to complete their tasks. By handing out more work at a time, there is more work done between each time this overhead occur.

5.1.3 OmpSs implementation using the CPUs and GPU with OpenCL

The fitness evaluation in genetic algorithms is independent for each individual. This means that highly parallel processors, like graphics processing units, can calculate fitness values for an large part of a population in parallel. OmpSs supports multiple implementations of tasks, and an implementation with both CPU and GPU tasks for running the fitness evaluation was used to explore the energy efficiency of this. OmpSs runs either the CPU or the GPU implementation of the tasks, based on the performance gain of running on the GPU and the overhead of switching to the GPU.

The island model for genetic algorithms used in the pure CPU implementation is not used in the OpenCL implementation. The reason is that there is only a single GPU, and the program needs to synchronize all CPUs every generation to do the fitness evaluation. With synchronization in every generation, the advantages of the island model are all gone.

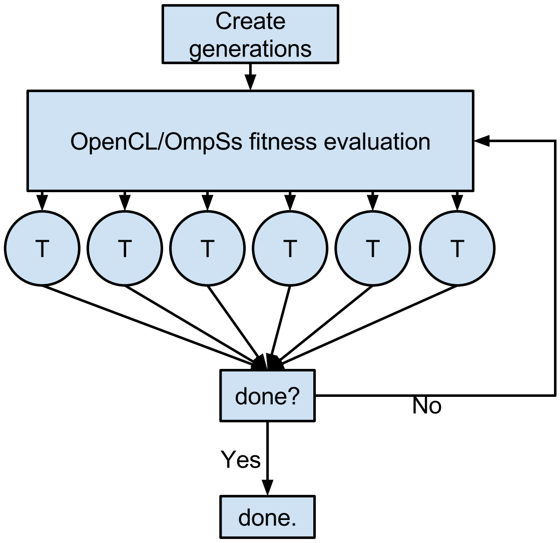


Figure 5.3: The program flow of the OmpSs and OpenCL genetic algorithm test application.

Figure 5.3 shows the program flow of the OmpSs genetic algorithm implementation using OpenCL. The OpenCL implementation works like the pure CPU implementation, but it is altered to utilize the GPU. To achieve this, the subpopulations and islands are gone in favor of of less synchronization. The entire population is evaluated in one OpenCL task. The evaluation, selection and mutation are still running in parallel as CPU OmpSs tasks.

The algorithm will first create a random population. When the population is ready, the program will start running generations. Each generation has three steps.

First the current population needs to have it's fitness values calculated. This implemented as both a CPU and GPU task, and OmpSs will select which one to run. The chosen task implementations will evaluate the fitness value of each individual.

When the fitness evaluation is completed, the next generation of individuals is generated. To utilize all the processor cores, one OmpSs task for each available core is created, each generating a part of the next generation. In Figure 5.3, there are six available processor cores, and six tasks creating a sixth of the next population are created.

The last step in each generation is to check if the conditions for ending the genetic algorithm are met. Either a sufficiently good individual has to be found, or the maximum number of generations has been reached. If the conditions are not met, the program will start another generation.

5.2 Cholesky decomposition

The implementation of the Cholesky decomposition application is from the Barcelona Supercomputing Center's example applications [29]. It was written for Intel based experiment platforms, as they used Intel Math Kernel Library (mkl). For usage in this thesis, the application was modified to be able to run it on the ARM based Odroid-XU3 platform. Calls to mkl were replaced with equivalent calls to two compatible libraries CLAPACK and CBLAS [30]. No further changes were made to the Cholesky decomposition application. Both LAPACK and BLAS were compiled and optimized for the test platform to achieve high performance.

The Cholesky decomposition applications implement the formulas from Equation 2.7 and Equation 2.8. A simple way to run this as an OmpSs application is to do the calculation of each item as an OmpSs task. As seen in the formula for the items, they have many dependencies. This can be implemented in OmpSs as input dependencies, and each task must have the item it calculates as an output dependency.

In the application used in this thesis tiles of items are used. Each tile is a sub matrix of A, with some tile size ts that is sent to the application when it is run. Tiles are used because of the overhead of running too many small tasks. When OmpSs tasks require just a small amount of processor time, the processor time spent on scheduling will be significant enough to affect the overall performance. By assigning the calculation of several matrix items belonging to a tile as a single OmpSs task, the overhead of scheduling the task occur fewer times. Each task will have the dependencies of all items in the tile, and when all dependencies are resolved the tile is put in the ready queue.When all tiles have been completed, the L matrix is done. Four linear algorithms, potrf, trsm, gemm and syrk, are used. The linear algorithms are available in the LAPACK and BLAS math libraries. Listing 5.1 shows the linear algorithms with the OmpSs annotations used, as well as the basic algorithm calling on them in function cholesky_blocked.

Listing 5.1: The tiled Cholesky decomposition algorithm.

```
#pragma omp task inout([ts][ts]A)
void omp_potrf(double * const A, int ts, int ld) { ... }

#pragma omp task in([ts][ts]A) inout([ts][ts]B)
void omp_trsm(double *A, double *B, int ts, int ld) { ... }

#pragma omp task in([ts][ts]A) inout([ts][ts]B)
void omp_syrk(double *A, double *B, int ts, int ld) { ... }
```

```c
#pragma omp task in([ts][ts]A, [ts][ts]B) inout([ts][ts]C)
void omp_gemm(double *A, double *B, double *C, int ts, int ld) { ... }

void cholesky_blocked(const int ts, const int nt, double* Ah[nt][nt])
{
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    omp_potrf (Ah[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      omp_trsm (Ah[k][k], Ah[k][i], ts, ts);
    }
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        omp_gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);
      }
      omp_syrk (Ah[k][i], Ah[i][i], ts, ts);
    }
  }
  #pragma omp taskwait
}
```

Chapter 6

Result and Discussion

This chapter covers the results from the two test applications. The genetic algorithm is run with several different configurations of the system using the small and big processor cores as well as the GPU. The Cholesky decomposition application is run with several configurations of the system using the small and big processor cores, but not the GPU. All the results are discussed with focus on what causes the results and what conclusions can be drawn from them.

6.1 Genetic algorithm

This section cover the results and discussion of the genetic algorithm application.

6.1.1 Serial implementation

To explore the overhead of running OmpSs, a simple serial genetic algorithm implementation was compared to the island model genetic algorithm implementation using OmpSs. The experiment was run with one ARM Cortex-A7, four ARM Cortex-A7s, and all four ARM Cortex-A7s and ARM Cortex-A15s.

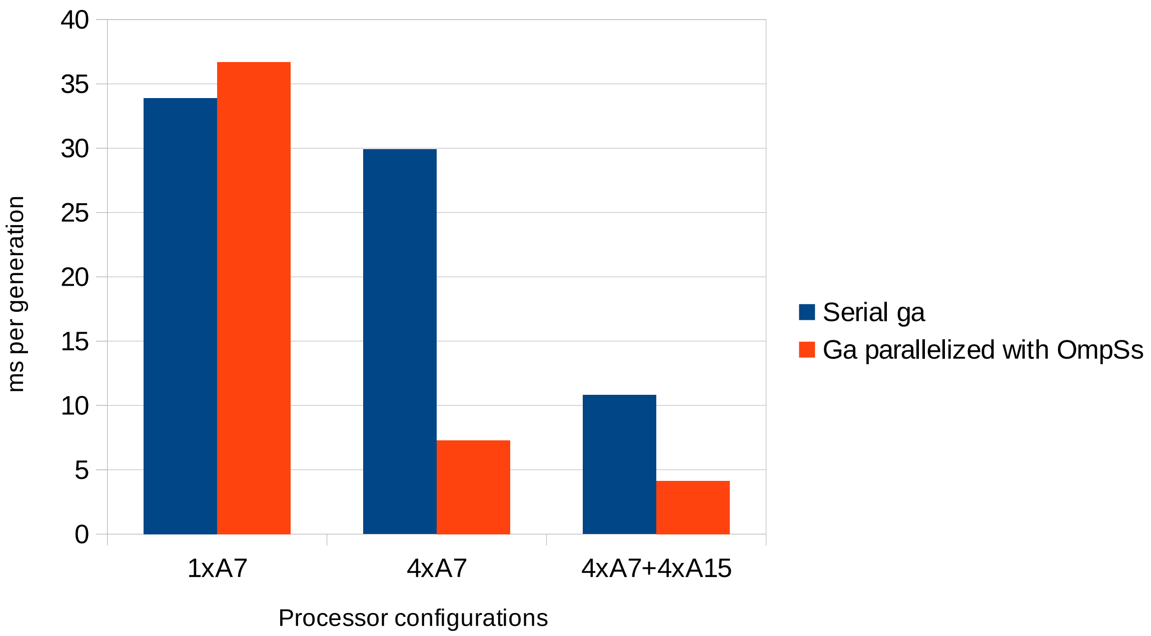


Figure 6.1: Execution time comparison between the plain C serial GA implementation and the island style parallel OmpSs implementation.

Figure 6.1 compares the results of the serial genetic algorithm implementation and the parallel OmpSs implementation. When running only a single ARM Cortex-A7, the OmpSs tasks can not run in parallel. This mean that the serial implementation has higher performance because of the time spent by OmpSs. The overhead caused by the runtime environment is 8.3%.

When our ARM Cortex-A7 cores are available, the OmpSs version can run tasks in parallel. The performance of the OmpSs version is four times higher than the serial version. This shows that OmpSs can parallelize a program efficiently. It is also worth noting that the serial version is experiencing a speedup of 1.13. This is because there are multiple processors available for other processes like the energy data collector and the operating system.

When the ARM Cortex-A15 cores are available there is a large speedup for both the serial and parallel implementations. The serial implementation can run on an ARM Cortex-A15 core. This gives a speedup of 2.8 compared to running with four ARM Cortex-A7 cores available. The parallel version is also experiencing a speedup when the number of available cores is increased. It is 2.6 times faster than the serial version.

6.1.2 OmpSs

This section covers the results of the genetic algorithm application running on the eight CPU cores using OmpSs.

Utilizing heterogeneity

Experiments with different configurations for the islands in the island model genetic algorithm were run. Figure 6.2 shows the two most interesting configurations.

Running one island per core was the most obvious strategy to run. With this configuration each OmpSs worker will run it's own island. The first four results running 1-4 ARM Cortex-A7 processor cores are scaling well with the added cores. When making the application run on a heterogeneous set of cores by making the ARM Cortex-A15 cores available, there is a sublinear speedup. This is because the ARM Cortex-A15 cores are completing their tasks a lot faster than the slower ARM Cortex-A7 cores. During each epoch of the genetic algorithm, there will be lost processing time on the large cores.

A solution to this is to divide the genetic algorithm into more tasks. This is done by creating two islands for each available core. Each time a worker complete a task, it will get a new one from the task queue. This allows OmpSs to balance the work by assigning more tasks to the large cores. The tasks from each island are run in parallel distributed out between large and small cores. The tasks are executed in a breadth first order, which makes all tasks in an epoch able to complete at about the same time. This eliminates the issue of fast cores spending much time waiting for slower cores to complete their work. Figure 6.2 shows that this island configuration greatly increase the performance scaling when adding high-performance cores.

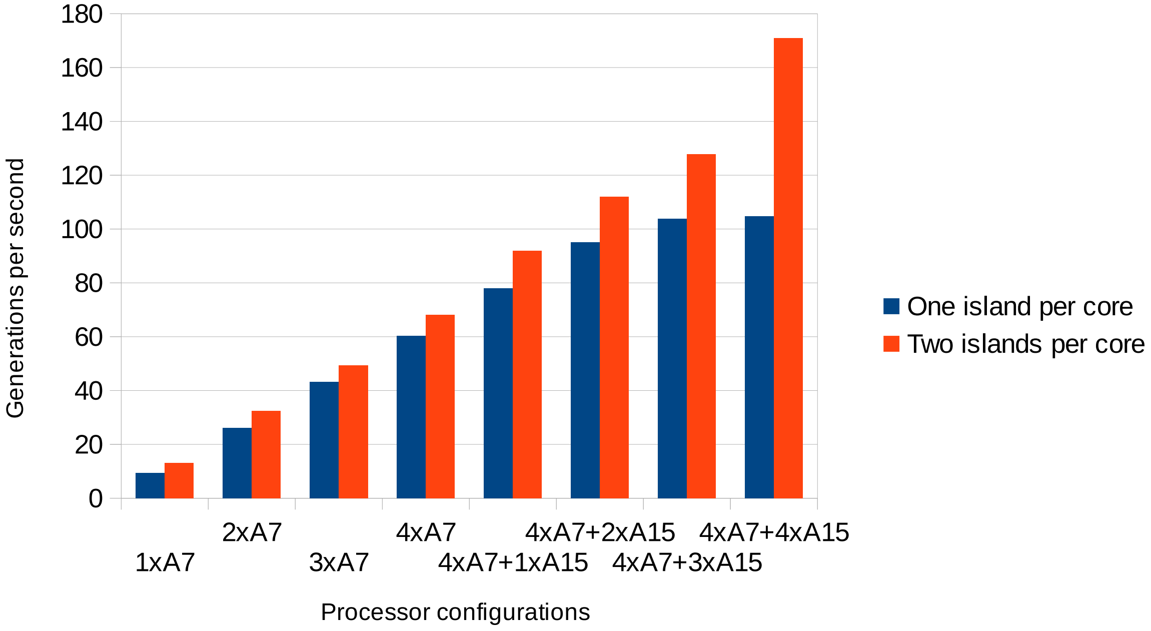
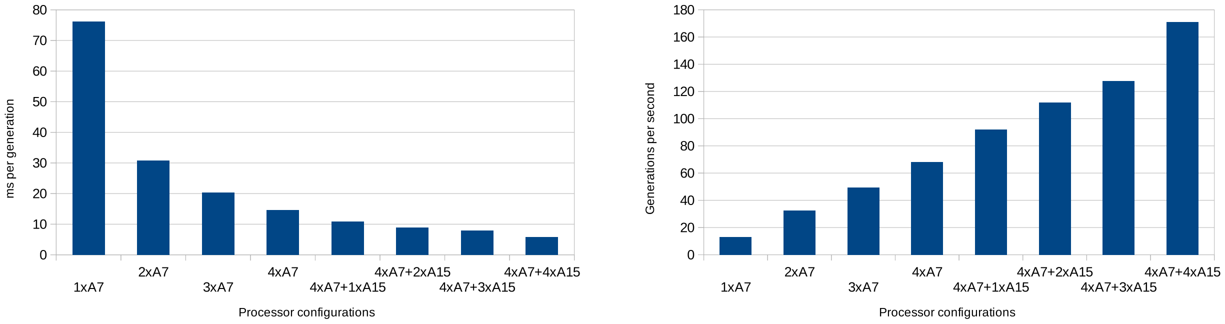


Figure 6.2: Filler

There are other possible alternatives for utilizing the heterogeneous processor that were not explored in this thesis. It is possible to adjust the population size on each island to assign a larger amount of work to the high-performance cores. This could be done by either testing different configurations and finding suitable ones or dynamically adjust the populations based on the islands current performance. It is also possible to have different epoch length for the islands run by the high-performance cores. This would be able to utilize the Bottom level-aware scheduler, discussed in Section 2.3, to make sure the large cores are assigned the islands with the longest epochs. Suitable sizes for the epochs can either be found by running experiments ahead of time or dynamically. Other strategies for utilizing the heterogeneity can most likely be found.

Because it had the best results, two islands per core is used for the rest of the thesis.

Performance



(a) Average execution time for each generation for different processor configurations.

(b) Average generations per second for different processor configurations.

Figure 6.3: The performance of the island model genetic algorithm running in parallel using OmpSs with different processor configurations. The graphs present the same data in two different ways. Figure a) is included to show the relation between the performance and the energy data in Figure 6.4, Figure 6.5 and Figure 6.6. Figure b) shows the performance scaling of the parallel application.

Figure 6.3 shows the performance of the island model genetic algorithm running with different configurations of the processor cores.

An interesting result here is that the performance gain of going from one to two ARM Cortex-A7 cores is more than doubled. The performance increase by adding a single core is 147%, which is a lot better than a linear scaling that would only yield an increase of 100%. The super linear performance scaling is also present when adding the two next cores, increasing performance with 52% and 38% respectively, opposed to the 50% and 33% increase expected with linear scaling. A likely reason for this super linear scaling is that there are other processes running on the system. It is also possible that the extra cache introduced by the other cores decrease cache misses, improving performance even more. The energy measurements are running in the background, as well as all actions done by the operating system. By spreading the work load on more processor cores, these other processes require a smaller percentage of the systems available processing power. The performance scaling well also show that OmpSs is parallelizing the program in an efficient way without a large overhead.

When the high-performance ARM Cortex-A15 cores are turned on, the performance keep scaling well. The super linear scaling of performance is not surprising here, as processing power is increased faster than the number of processors is increased. It shows that OmpSs can utilize the fast processor cores by assigning more tasks to them. This way of utilizing the heterogeneity of the Samsung Exynos 5 system on a chip was the goal of this application. Running on all eight processor cores is 2.5 times faster than running on only the small cores.

Power

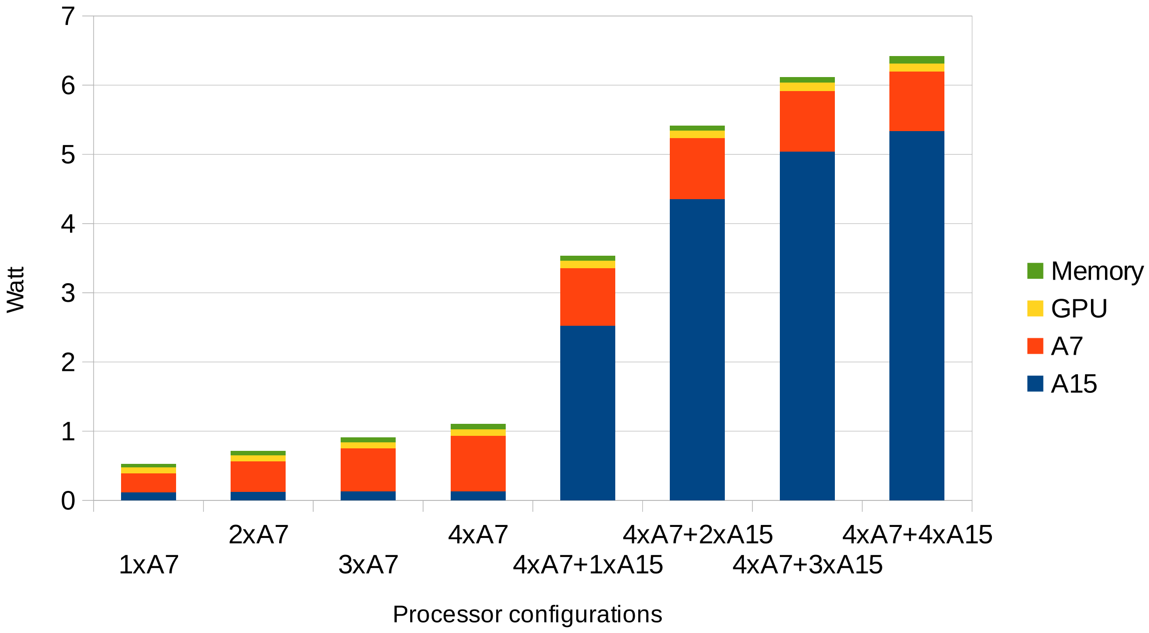


Figure 6.4: Average power consumption while running the island model genetic algorithm application on the CPU with different processor configurations. The power consumption of the four ARM Cortex-A7 cores, the four ARM Cortex-A15 cores, the ARM Mali-T628 and the memory is shown.

Figure 6.4 shows the power consumption of the components of the system with energy sensors available. The measurements are the average values while running the island style genetic algorithm with different processor configurations.

As expected the lowest power consumption is found when all but one energy efficient ARM Cortex-A7 core is turned off. This mean that if the system needs to be left on without performing any particular task, this configuration is the most effective. This is useful for systems like mobile phones that are left idling for long periods because the user want long battery life.

As expected the measurements show that the power increase when more cores are used. They also show that the power does not increase linearly when adding more cores. For each ARM Cortex-A7 core that is turned on, there is an increase of power, but the average power per core goes down. The power of running on all four ARM Cortex-A7 cores is 2.9 times the power of running on only one. Also, the other components of the system are also drawing power. The measurements show that the power consumption of the ARM Cortex-A15 cores, ARM Mali-T628 GPU, and the memory is not affected by the number of ARM Cortex-A7 cores that are used. The power increase of running on four cores is 111% compared to running on only one.

Turning on an ARM Cortex-A15 core drastically increase the power consumption of the system. There was a power consumption by these cores when they where off, but turning on one of the cores increase the power 18.9 times. Turning on more of the ARM Cortex-A15 cores increase the power consumption further, but not as much as turning on the first one. This shows that the processor cores share parts of the system that will draw power some amount of power when one or more of the cores are used. It can be more efficient to run on all of the large cores for a short period before turning them off, rather than just using some of them.

When varying the number of ARM Cortex-A15 cores, the power consumption of the ARM Cortex-A7, ARM Mali-T628, and the memory is not affected.

Energy

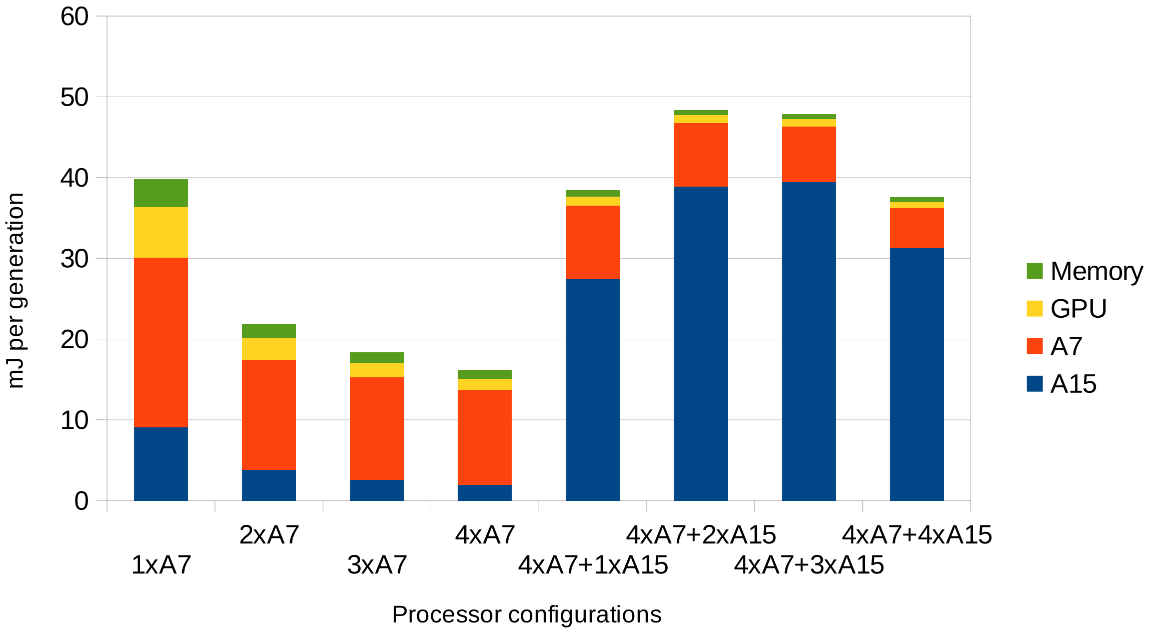


Figure 6.5: Average energy consumption of each generation while running the island model genetic algorithm application on the CPU with different processor configurations. The energy consumption of the four ARM Cortex-A7 cores, the four ARM Cortex-A15 cores, the ARM Mali-T628 and the memory is shown.

Figure 6.5 shows the average energy consumption of executing a single generation of the island style genetic algorithm experiments. The energy efficiency of the experiment is shown through these data. These results are the amount of energy required to run the application for each different processor configuration. The results are calculated as the product of the execution time from Figure 6.3 and the power from Figure 6.4.

Both the power consumption and performance scaled very well with more ARM Cortex-A7 cores added. Because of this we also see that the energy efficiency get better when more of them are added. As can be seen from the size of the other components energy consumption, the unused big cores, GPU and memory represent a larger portion of the total energy consumption when running on fewer cores. This is because they are left drawing power for a longer time. The best energy efficiency for the system is found when running on the four ARM Cortex-A7 cores, with all the ARM Cortex-A15 cores turned off.

When the big cores are turned on, the energy consumption increase a lot. Using only one ARM Cortex-A15 core is consuming less energy than running on two or three ARM Cortex-A15 cores. This is because the energy efficient ARM Cortex-A7 cores are doing a lot of the work effectively. The best energy efficiency while running with ARM Cortex-A15 cores is when running all four of them, with a margin of 2% compared to running with one ARM Cortex-A15 core. Running on all eight cores consumes 2.3 times the energy of running on only the four small cores. For a user, it can be more important to have high performance than to save energy in many situations. In such situations, the user can increase energy consumption by 2.3, to gain the 2.5 times increase in performance found in Figure 6.3. This ability to choose between performance and energy efficiency as necessary is a strong argument for utilizing heterogeneous processors.

Energy delay product

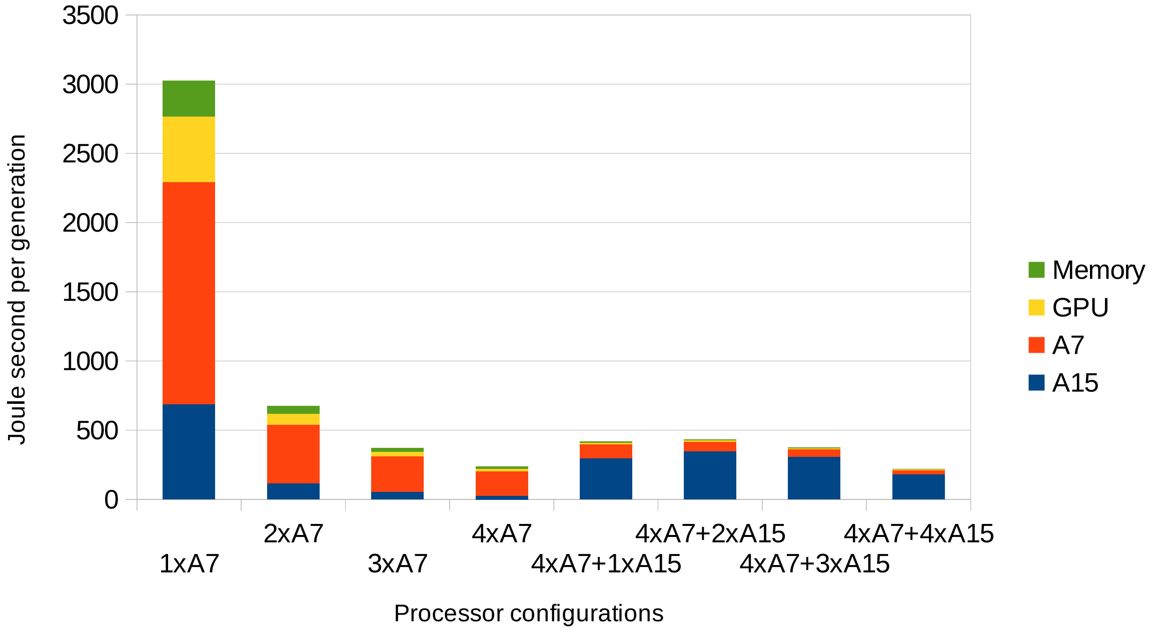


Figure 6.6: Average energy delay product of each generation while running the island model genetic algorithm application on the CPU with different processor configurations. The energy consumption of the four ARM Cortex-A7 cores, the four ARM Cortex-A15 cores, the ARM Mali-T628 and the memory is shown.

Figure 6.6 shows the energy delay product of the island style genetic algorithm experiments. It serves as a way to compare the performance trade off from running on the energy efficient processor configurations. The results are the product of the execution time from Figure 6.3 and the energy from Figure 6.5.

The results show that there are two feasible options with energy delay products really close to each other. The other processor configurations lack the energy efficiency or performance to compete with them. The two options are running on all four ARM Cortex-A7 cores and running on all eight ARM Cortex-A7 and ARM Cortex-A15 cores. The energy delay product of all eight cores is 8% higher than that of the four small cores. While running on all eight cores results in the best energy delay product, the close results show that there is a choice to be made. For applications where either performance or energy efficiency is more important than the other, different processor configurations would be optimal.

6.1.3 OmpSs with and without OpenCL

A GPU can run code highly parallel OpenCL code. In this section, dual implementations of the fitness evaluation step of the genetic algorithm are explored. OmpSs runs the fitness evaluation on either the GPU as an OpenCL implementation or the CPU as conventional C.

Performance

As discussed in Section 2.6.1, the Samsung Exynos 5422 feature a heterogeneous cooperations between the GPU and CPU. This allows running OmpSs tasks with multiple implementation that can run on the CPU or GPU from the same shared memory. A modified version of the OpenCL genetic algorithm, implemented with conventional OpenCL with data transfer over the bus was used to get explore to the efficiency of using the shared memory for data transfer. The genetic algorithm spent 2.1 seconds on memory operations and data transfer per generation. When increasing the fitness evaluation complexity, it scaled just as well as the OmpSs version using the shared memory for data transfer, but the large overhead of data transfer shows the advantage of the heterogeneous properties of the Samsung Exynos 5422.

It is worth noting that OmpSs was easy to get working with the dual implementations from a programmers point of view. The development of the dual implementation OpenCL code was faster than the development of the conventional OpenCL with data transfer code.

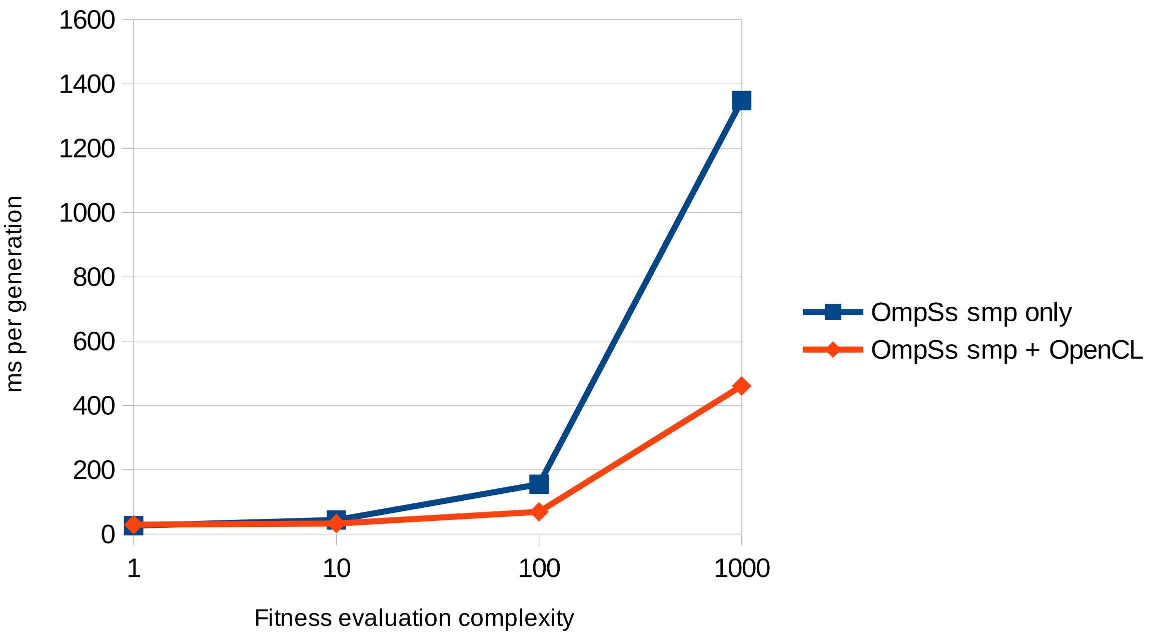


Figure 6.7: Average execution time for each generation while running the genetic algorithm with and without utilizing OpenCL. One implementation uses only a CPU task to run the fitness evaluation, while the other has dual OmpSs implementations for SMP and OpenCL.

Figure 6.7 shows the average execution time of each generation of the genetic algorithm. The results are from multiple genetic algorithm executions with a different computational complexity of the fitness evaluation. The fitness function used in the result on the far right require 1000 times more processor time than the one on the far left.

The results show that for the simplest fitness function, the CPU-only version of the genetic algorithm is 11% faster than the version utilizing dual implementations for the CPU and GPU. With this fitness complexity, the CPU is working faster than the GPU because there is an overhead of using the GPU. The rest of the results have more complex fitness functions that are executed faster on the GPU than on the CPU. For these results, the dual implementation version is performing better than the pure CPU version. The higher the complexity of the fitness evaluation, the higher the performance advantage of the dual implementation version gets. Around a fitness complexity of 1000 the scaling stabilized, and for larger complexities the dual implementation version was consistently about four times faster than the smp version.

Power and energy

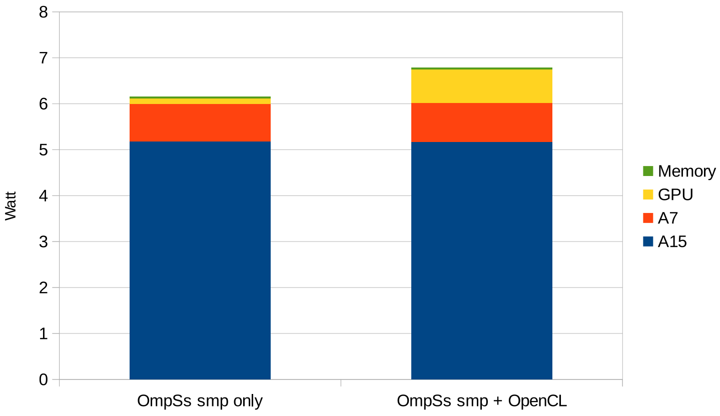
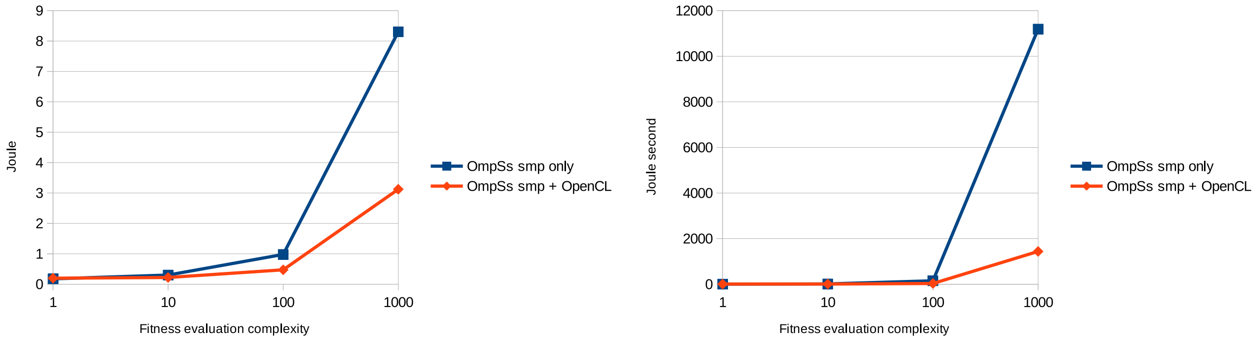


Figure 6.8: Average power for the ARM Cortex-A7 cores, ARM Cortex-A15 cores, ARM Mali-T628 GPU, and the memory while running the genetic algorithm with and without utilizing OpenCL. One implementation uses only a CPU task to run the fitness evaluation while the other has dual OmpSs implementations for smp and OpenCL.

Figure 6.8 shows the average power consumption of the genetic algorithm experiments using OpenCL. The data shown here are from the runs with a fitness complexity of 1000. The data was almost identical for all the different runs, but with significantly lower power for the GPU in the OmpSs + OpenCL implementation when the fitness complexity was lower. This is because more time is spent with the GPU idle when the fitness is not very complex, allowing the average power consumption to be lower.

With the exception of the GPU, all components draw the same power in both versions within an error of 4.3%. The only significant difference is that the GPU draws 5.9 times as much power when the OpenCL implementation is used. As the power consumption of the GPU is small compared to the rest of the system, there is not a large power overhead by using the GPU. The effect of this is evident in Figure 6.9.



(a) Average energy consumption while running the genetic algorithm with and without utilizing OpenCL.

(b) Average energy delay product while running the genetic algorithm with and without utilizing OpenCL.

Figure 6.9: Average energy consumption and energy delay product while running the genetic algorithm with and without utilizing OpenCL. One implementation uses only a CPU task to run the fitness evaluation while the other has dual OmpSs implementations for smp and OpenCL.

Figure 6.9a shows the energy consumption of the two implementations for different fitness evaluation complexities. The results are the product of the execution time from Figure 6.7 and the power from each run like the ones shown in Figure 6.8.

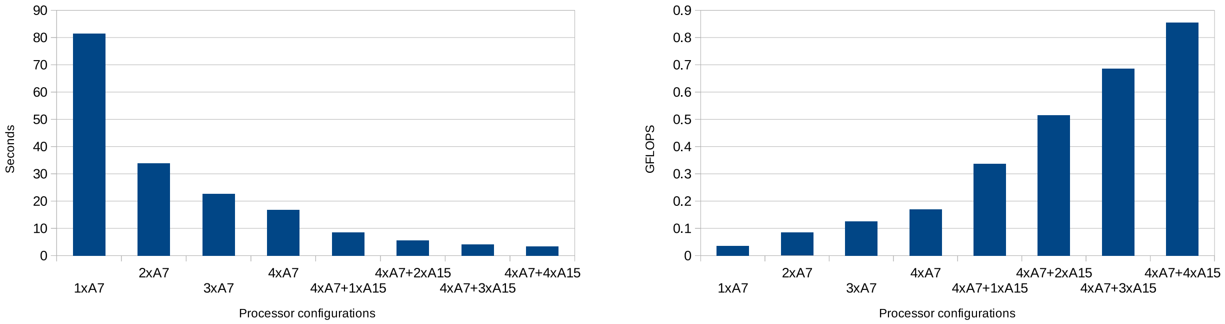
The consequence of the small variation in power is that performance is important. The energy consumption and energy delay product follow the same scaling as the performance. The smp only implementation uses less power, but the performance advantage of the smp + OpenCL implementation far overpowers this.

The results show that for many genetic algorithm problems where the fitness evaluation require significant processor time, utilizing the GPU is good for both performance and energy efficiency.

6.2 Cholesky decomposition

This section present and discuss the results from the Cholesky decomposition experiments. The Cholesky decomposition is divided into tasks by dividing the matrix into tiles that can be calculated in parallel as described in Section 5.2.

Performance



(a) Average execution time for the OmpSs 2048x2048 Cholesky decomposition for different processor configurations.

(b) Average GFLOPS for the OmpSs 2048x2048 Cholesky decomposition for different processor configurations.

Figure 6.10: The performance of the OmpSs 2048x2048 Cholesky decomposition running in parallel using OmpSs with different processor configurations. The graphs present the same data in two different ways. Figure a) is included to show the relation between the performance and the energy data in Figure 6.11, Figure 6.12 and Figure 6.13. Figure b) shows the performance scaling of the parallel application.

Figure 6.10 shows the performance of the Cholesky decomposition test application running on different processor configurations.

It is interesting to observe that the performance scaling when running on one to four ARM Cortex-A7 processors is super linear. The performance is increased by 140% when going from one to two processor cores rather than just being doubled. The performance increase from adding a third and fourth processor core is barely higher than linear. This is the same performance scaling as observed for the genetic algorithm. As discussed in the performance discussion in Section 6.1.3, this is caused by the extra processor cores sharing the load of running other tasks on the ODROID-XU3.

When the ARM Cortex-A15 cores are turned on, there is a large increase in performance. Each added core increase the throughput with close to 0.17 GFLOPS. This linear scaling shows us that the application is utilizing the available processing power well.

The performance scaling shows that each ARM Cortex A15-core contribute about as much processing power as the four ARM Cortex-A7 cores combined. This is a lot better than the scaling found while running the genetic algorithm experiment, where each big core contributed only

59% of the combined performance of the small cores. This shows that the Cholesky decomposition application utilizes the heterogeneous processor a lot better then the genetic algorithm. Other ways of parallelizing the genetic algorithm can possibly utilize the combination of big and small cores better.

Power

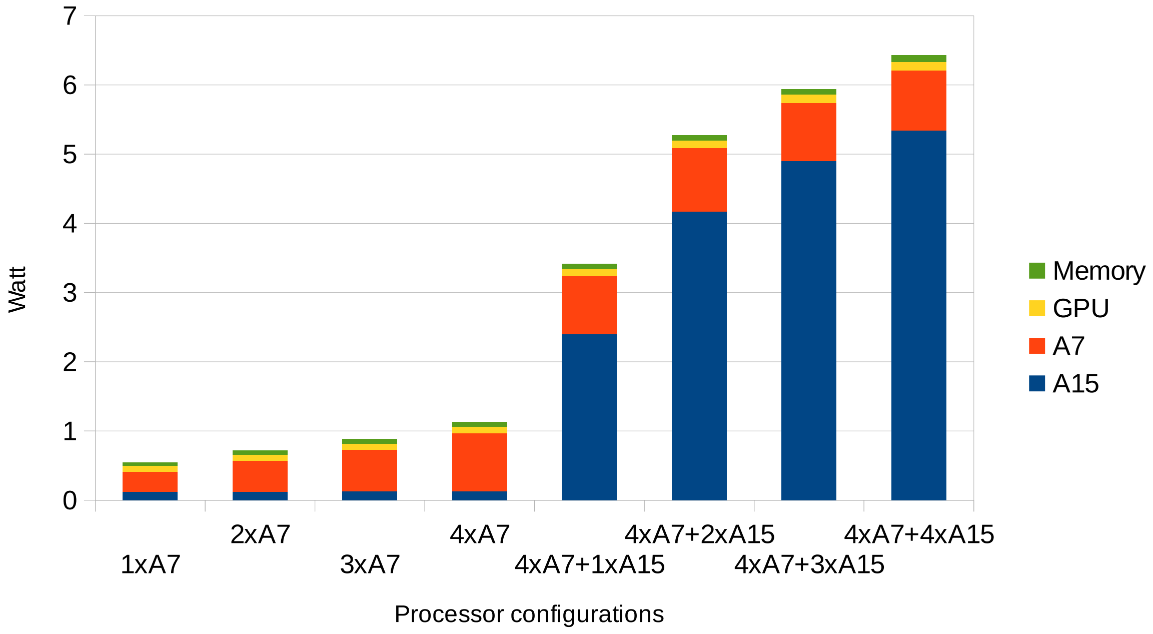


Figure 6.11: Average power consumption while running the 2048x2048 Cholesky decomposition application on the CPU with different processor configurations. The power consumption of the four ARM Cortex-A7 cores, the four ARM Cortex-A15 cores, the ARM Mali-T628 GPU and the memory is shown.

Figure 6.11 shows the power consumption of the components of the system with energy sensors available. The measurements are the average values while running the Cholesky decomposition experiments with different processor configurations. All the results are very close to the results found in the power discussion in Section 6.1.2. This shows us that the system has a very similar power consumption even though the tasks being run differ. See the discussion in Section 6.1.2 for an analysis of the power consumption.

Energy

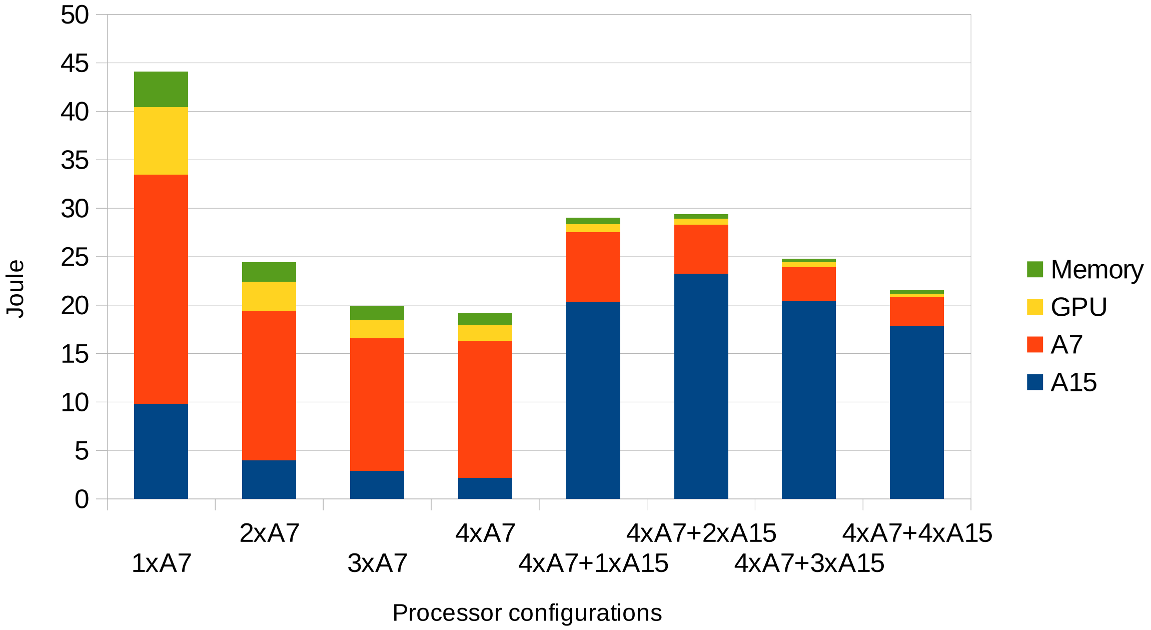


Figure 6.12: Average energy consumption of a full execution of the 2048x2048 Cholesky decomposition application on the CPU with different processor configurations. The energy consumption of the four ARM Cortex-A7 cores, the four ARM Cortex-A15 cores, the ARM Mali-T628 and the memory is shown.

Figure 6.12 shows the amount of energy spent on average executing the Cholesky decomposition application. The results are calculated as the product of the execution time from Figure 6.10a and the power from Figure 6.11.

The performance scaled linearly when adding more small cores and linearly when adding more big cores. The power of the system however, had a large overhead by turning on just some large cores or just some small cores. This affects the energy results, and cause there to be only two options that are viable to use for energy efficiency. Running on all four ARM Cortex-A7 cores is the most energy efficient option. This mean that if performance is not an issue, there is no reason to run any other processor configuration. Running all eight cores is less energy efficient than running on only the small cores, consuming 17% more energy. The difference is not great, and situations can occur where the small decrease in energy efficiency can be worth it for the increase in performance. Because of this, running on all eight cores can't be dismissed as a feasible compromise between energy and performance.

It is worth noting that these are the same processor configurations that were found to be viable options for running the genetic algorithm application. The Cholesky decomposition application has a lot closer energy results for the big and the small cores. This is because of the way the Cholesky decomposition application had better performance scaling with the big cores.

Energy delay product

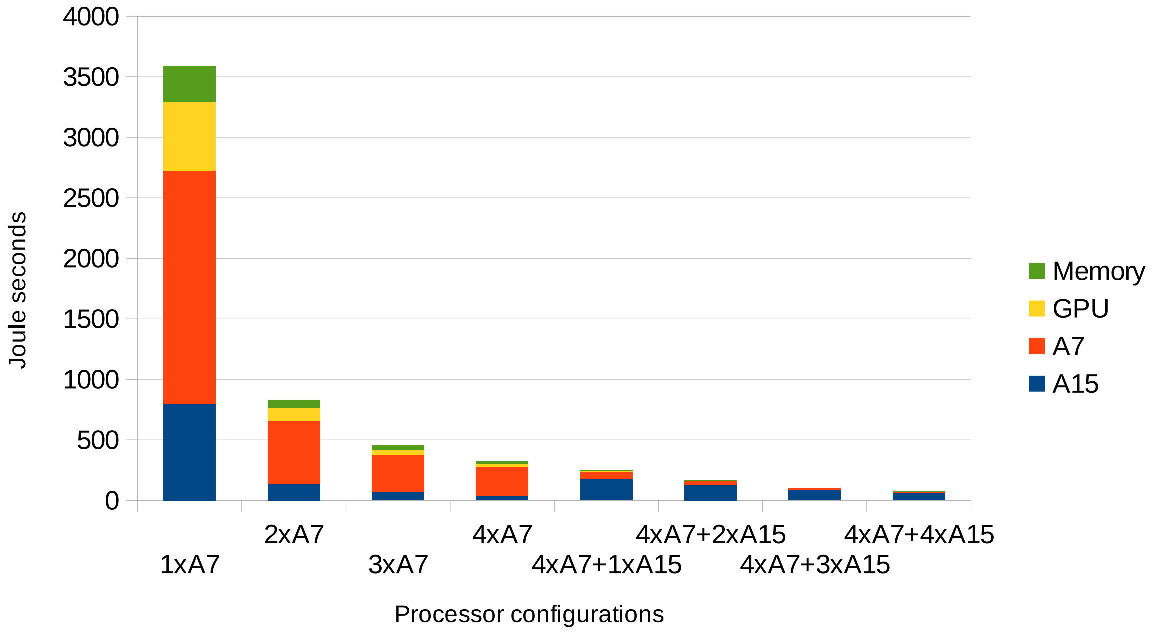


Figure 6.13: Average energy delay product of a full execution of the 2048x2048 Cholesky decomposition application on the CPU with different processor configurations. The energy consumption of the four ARM Cortex-A7 cores, the four ARM Cortex-A15 cores, the ARM Mali-T628 and the memory is shown.

Figure 6.13 shows the energy delay product of running the Cholesky decomposition application. It serves as a way to compare the performance trade off from running on the energy efficient processor configurations. The results are calculated as the product of the execution time from Figure 6.10a and the power from Figure 6.12.

The results shows that for each added core, the energy delay product is better. This shows that the energy efficiency advantage of the ARM Cortex-A7 cores is not great enough to overcome the performance increase from utilizing the ARM Cortex-A15 cores. When running the Cholesky decomposition application, all eight cores should be used if a balance between performance and energy efficiency is the objective.

The results are significantly different from the genetic algorithm application. For the genetic algorithm, the energy delay product of running on the four small cores was close to that of running on all eight. The difference is caused by the large increase in performance when the large cores are used in the Cholesky decomposition application. It is still interesting to see that for different application different processor configurations can be chosen under some circumstances.

Chapter 7

Conclusion

In this thesis, the energy efficiency and performance of OmpSs task based programming on heterogeneous systems was evaluated. Experiments with a genetic algorithm application and a Cholesky decomposition application were run on an ODROID-XU3 featuring a Samsung Exynos 5 heterogeneous multiprocessor. The experiments were run on different combinations of energy efficient ARM Cortex-A7 processor cores, high performance ARM Cortex-A15 processor cores and a ARM Mali-T628 GPU. The results show that it the ability to vary what processors to use can be effective in achieving different goals for different applications. For applications with large enough parallel parts, GPGPU can be utilized for a dramatic increase in both performance and energy efficiency. For other applications execution on small energy efficient cores is preferable over high performance cores when energy efficiency is the most important. When performance is an objective it is possible to trade energy efficiency for increased performance. The results showed that OmpSs task based programming is a feasible model for programming heterogeneous systems. The overhead of running OmpSs was found to be small, and task based programming was able to abstract the parallelization of the application from most of the development. The ability to dynamically run tasks with multiple implementations on either a GPGPU or a CPU is a big advantage of OmpSs, allowing programs with this complex parallelization to be implemented much simpler.

7.1 Performance and energy efficiency

The small ARM Cortex-A7 cores deliver just a fourth of the performance of the big ARM Cortex-A15 cores. The big cores does however consume 6.6 times more power. This mean that it is possible to trade energy efficiency for performance on such systems as the Samsung Exynos 5. Applications with performance as the primary objective can run on all eight ARM Cortex-A7 and ARM Cortex-A15 cores. Applications with energy efficiency as the primary objective can run on the four energy efficient ARM Cortex-A7 cores. This way it is possible to develop dynamic systems that choose what processors to utilize based on needs.

Multiple implementations of OmpSs tasks allow for utilization of GPGPUs. The performance increase from using the GPGPU is large for many applications. The Samsung Exynos 5422 uses the GPGPU as a special core in the heterogeneous system, resulting in a low overhead of using the GPU. This great performance increase together with the low power of the GPU gives it excellent energy efficiency. Applications can run with better energy efficiency using OmpSs tasks running on a GPU.

7.2 Heterogeneous multi-processors and the task based programming model

The complexity of developing parallel programs for heterogeneous systems is high for conventional development. OmpSs is a proposed model for lowering this complexity by using the scheduler to dynamically manage the parallel execution. The scheduler is able to manage tasks and their dependencies and execute the applications on the available workers. The use of small and big cores is not an issue the programmer have to address since this task can be solved dynamically by the scheduler. Tasks also support multiple implementations for different devices. Creating programs that dynamically run in parallel on both available CPUs and GPUs is complex and require much special purpose code, OmpSs creates an abstraction from this task and allow this kind of development to be much easier. The performance and energy overhead of running OmpSs is low, and for many purposes the dynamic parallelization can give better results. Task based programming with OmpSs was found to be a feasible solution to implementing parallel programs on heterogeneous systems.

Bibliography

[1] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, et al. The hipeac vision. *Report, European Network of Excellence on High Performance and Embedded Architecture and Compilation*, 12, 2010.

[2] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

[3] Rune Holmgren. Energy efficiency experiments on exynos 5 using ompss. 2014.

[4] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[5] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[6] Easy programming heterogeneous systems. http://www.cs.vu.nl/complexhpc2011/slides/complexHPC2011-StarSs.pdf, 2011.

[7] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 4.0 edition, 7 2013.

[9] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[10] Nanos++ runtime library. https://pm.bsc.es/projects/nanox. accessed: 2015-05-14.

[11] Encore. http://www.encore-project.eu/about. Accessed: 2015-05-15.

[12] Barcelona Supercomputing Center. Ompss user guide. 2014.

[13] Ashish Venkat and Dean M Tullsen. Harnessing isa diversity: design of a heterogeneous-isa chip multiprocessor. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 121–132. IEEE, 2014.

[14] Peter Greenhalgh. big.little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8, 2011.

[15] Arm.com. http://www.arm.com/products/processors/technologies/biglittleprocessing.php. Accessed: 2015-02-25.

[16] Brian Jeff. big.little technology moves towards fully heterogeneous global task scheduling. 2013.

[17] Hardkernel.com. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127&tab_idx=2. Accessed: 2014-10-25.

[18] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[19] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[20] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

[21] William H.; Saul A. Teukolsky; William T. Vetterling; Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (second edition).* Cambridge University England EPress, 1992.

[22] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal*, 2006(142):7, 2006.

[23] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[24] Hallgeir Lien, Lasse Natvig, Abdullah Al Hasib, and Jan Christian Meyer. Case studies of multi-core energy efficiency in task based programs. In *ICT as Key Technology against Global Warming*, pages 44–54. Springer, 2012.

[25] Kit-Sang Tang, Kim-Fung Man, Sam Kwong, and Qun He. Genetic algorithms and their applications. *Signal Processing Magazine, IEEE*, 13(6):22–37, 1996.

[26] Mohammad Hosein Sedaaghi, Constantine Kotropoulos, and Dimitrios Ververidis. Using adaptive genetic algorithms to improve speech emotion recognition. In *Multimedia Signal Processing, 2007. MMSP 2007. IEEE 9th Workshop on*, pages 461–464. IEEE, 2007.

[27] Alan George, Michael T Heath, and Joseph Liu. Parallel cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and its applications*, 77:165–187, 1986.

[28] John leflohic. `http://www-cs-students.stanford.edu/~jl/Essays/ga.html`.

[29] Programming models @ bsc. `http://pm.bsc.es/`.

[30] Netlib clapack. `http://www.netlib.org/clapack/`. Accessed: 2015-04-12.