Benchmarking novel multi-SoC DSP architecture

Sioutas, T.

Award date:
2015

Technische Universiteit Eindhoven University of Technology

PRODRIVE TECHNOLOGIES

5T746

Benchmarking novel multi-SoC DSP architecture

Thomas Sioutas
(0869715)
t.sioutas@student.tue.nl

University Supervisors
prof. dr. Henk Corporaal
h.corporaal@tue.nl

Gert-Jan van den Braak
g.j.w.v.d.braak@tue.nl

Prodrive Supervisor
Nikos Larisis
nikos.larisis@prodrive-technologies.nl

Eindhoven, 30 November 2015

Contents

Abbreviations

AMC	Advanced Mezzanine Card
APBC66	AMC Piggy Back C66x
API	Application Programming Interface
ARM	Acorn RISC Machine
ASET	Algorithmic Species Extraction Tool
ATCA	Advanced Telecommunications Computing Architecture
6PU_BLADE	6 Processing Unit Blade
BM3D	Block Matching 3D
BW	Bandwidth
CRF	Critical Request Flow
DDR3	Double Data Rate type three
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
ECC	Error-Correcting Code
GPU	Graphic Process Unit
I^2C	Inter-Integrated Circuit
ILP	Instruction Level Parallelism
MACS	Multiply-Accumulates Per Second
MCA	Modular Component Architecture
MCSDK	Multicore Software Development Kit
MIPS	Microprocessor without Interlocked Pipeline Stages
MIPS	Million Instructions per Second
MOPS	Million Operations per Second
MPM	Multiple Processor Manager
MSMC	Multicore Shared Memory Controller
NL-M	Non Local Means
NUMA	Non-Uniform Memory Access
OFED	Open Fabrics Enterprise Distribution
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OpenMPI	Open Message-Passing Interface
PDAK2H	Prodrive DSP AMC Keystone-II Hawking
PSNR	Peak Signal-to-Noise Ratio
PU	Processing Unit
RIO	RapidIO
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SoC	System on Chip
SRIO	Serial RapidIO
TI	Texas Instruments
TT	Transport Type

Abstract

Advances in real-time constrained embedded software applications lead to continuous research and development of multi-SoC heterogeneous platforms. A considerable amount of research has been put towards the development of clusters of autonomous heterogeneous multi-SoC platforms which can provide up to thousands of processing cores. Yet, benchmarking a big heterogeneous cluster and identifying its bottlenecks is not always feasible since it is an expensive and time consuming procedure. However, analytical performance models can be used. By introducing an abstract description of a complicated system with mathematical equations they provide a prediction about the capabilities and the performance of the system, they identify performance bottlenecks and help programmers tune their applications without exploring all the potential configurations of the system. The main objective of this thesis is the deduction of analytical performance models for DSP heterogeneous platforms. For the experimental validation of the models, benchmarking on two different proprietary heterogeneous multi-SoC DSP platforms was conducted. Moreover, a set of guidelines for selecting a target platform based on an application's characteristics is defined and a methodology to distribute optimally an application over a distributed platform is introduced.

1. Introduction

Medical, automotive, defence and aerospace are only some examples of markets that use real-time constrained embedded software applications. These examples give a good estimate of the great importance of embedded applications in everyday life. However, in order to satisfy their high performance requirements the need of continuous research and development of platforms that are capable to support flawless execution of such applications increases constantly.

Despite of the great research that is made this far, the physical limits of semiconductor-based microelectronics introduce important restrictions on the CPU performance. The above have led to the development of multi-core platforms (Figure 1.1) that are able to offer greater performance than single-core platforms and meet the requirements of high demand applications.

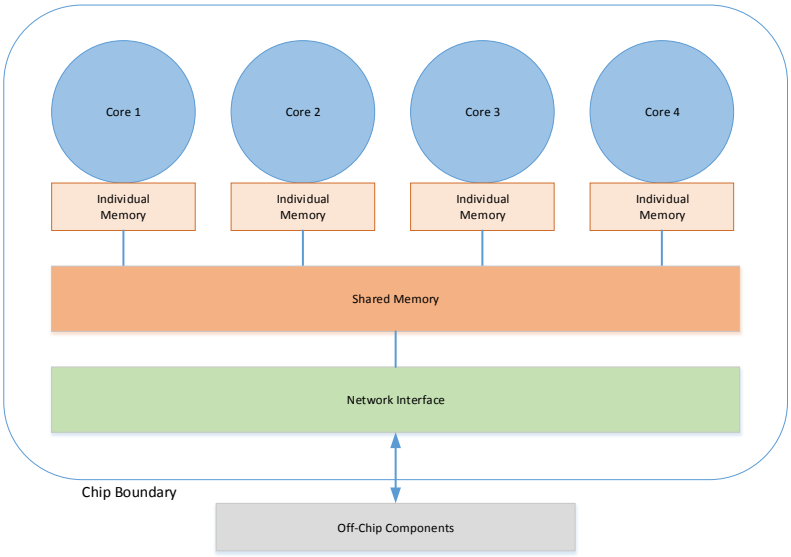

Figure 1.1 Multi-core platform abstraction.

However, as the number of cores integrated on a chip increases the on-chip power consumption becomes a critical issue. Heterogeneous cores on a chip is an alternative approach for multi-core platforms. It is a promising solution not only for increasing compute performance but also for power-efficient computing. Platforms consisted by multiple heterogeneous multi-core SoCs (Figure 1.2) gain performance not only by adding cores, but also by incorporating specialized processing capabilities to handle particular tasks.

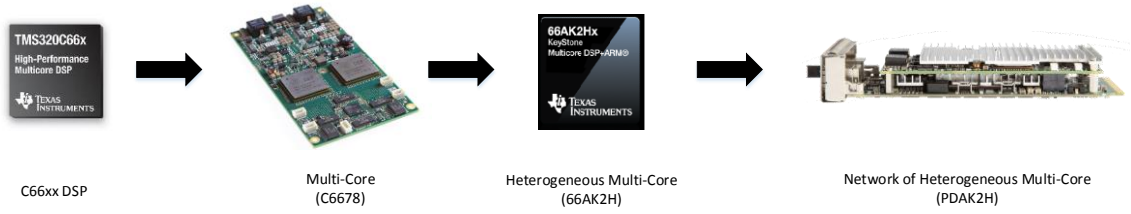

Figure 1.2 Multi-core SoC to Multi-SoC heterogeneous platform evolution.

However, selecting the most suitable target platform for a specific applications is not a simple procedure. There is a significant number of issues that need to be taken into account.

1.1. Problem Statement

By the process of benchmarking it is possible to investigate the typical issues that have to be solved in a multi-SoC heterogeneous platform:

- The scalability issues of the platform
- Communication (bandwidth, latency, overhead, etc.) between cores and between cores and memory
- Selection of the system interconnection (Figure 1.3)
- Development usability due to differences between the traditional programming approaches for multicore CPUs and DSPs/GPUs

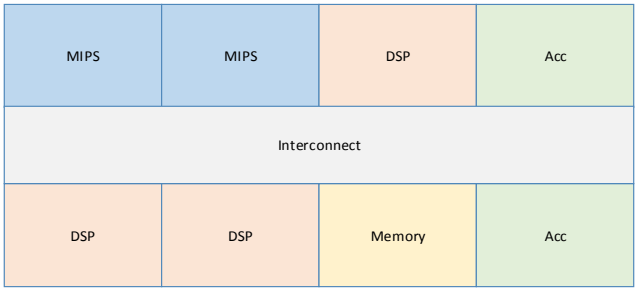

Figure 1.3 System Interconnect.

Moreover, a considerable amount of research has been put towards the development of clusters of autonomous heterogeneous platforms which can provide up to thousands of processing cores. However, benchmarking a big heterogeneous cluster is a very time consuming procedure and programmers are not always able to proceed into an exhaustive design space exploration of each available target platform configuration. In this case, analytical performance models are used. By providing a prediction about the capabilities and the performance of the system, performance bottlenecks can be identified and help programmers tune their applications without exploring all the potential configurations of the system.

Hence, after setting the requirements that need to be met by the application (e.g. execution time), the programmer can select via the analytical performance models the target platform which is able to achieve the required performance.

The next step after specifying the capabilities as well as the performance bottlenecks of the target platform is to develop the application that has to be ported. In heterogeneous platforms, this procedure can be challenging because of the differences not only in the architecture of the available processing units but also in the programming methods that are used. Moreover, the code that has been written for a single heterogeneous platform has to be rewritten in order to be ported into a heterogeneous cluster. Hence, extra effort is needed by the programmer in order not only to rewrite the code but also to proceed to an efficient resource management.

1.2. Thesis objective and contributions

Concerning the aforementioned issues, the main objectives of this thesis are:

- The deduction of analytical performance models for DSP heterogeneous platforms
- The definition of a set of guidelines for selecting a target platform based on an application's characteristics
- The introduction of a methodology to optimally distribute an application over a heterogeneous platform

For experimentally validating the thesis objectives, benchmarking on two different proprietary heterogeneous multi-SoC platforms (Figure 1.4) was conducted. Prodrive Technologies develops PDAK2H (Figure 1.4 (a)), a computer mezzanine card which is assembled with the TI Keystone-II "Hawking" SoC which offers in total 4 ARMs and 8 DSPs. Benchmarking of TI Keystone-II "Hawking" SoC was performed during the preparation phase of the thesis project along with an investigation regarding three different programming methods. The introduction of two additional SoCs, offering in total 16 extra DSPs, on PDAK2H is feasible. The contribution of the extra 16 DSPs in the PDAK2H was verified during the final phase of the thesis project. Moreover, for the

final phase, the benchmarking of the Prodrive ATCA-TK2-6PU blade was made. The Prodrive ATCA-TK2-6PU blade is assembled with 6 PUs containing per PU: one TI “Keystone-II” (66AK2H14) and two “Keystone-I” (TMS320C6678) SoCs offering four ARM cores and 24 DSP cores.

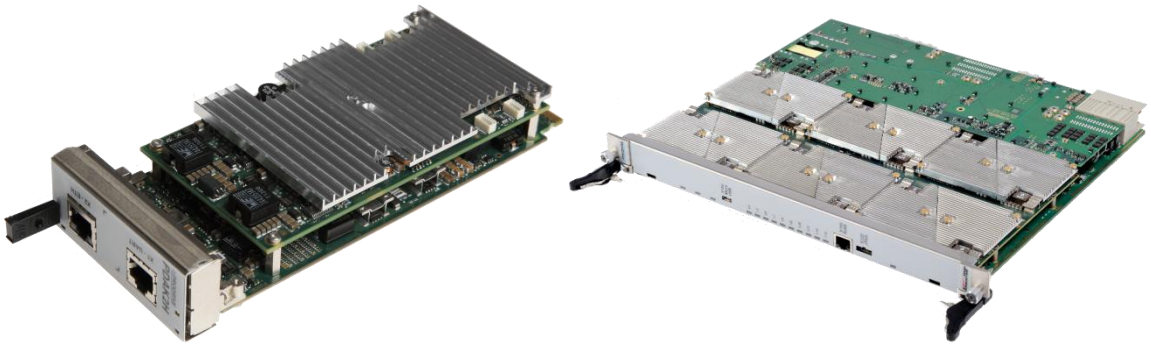

Figure 1.4 (a) PDAK2H [4], (b) ATCA-TK2-6PU Blade [6].

Also, an analysis regarding the behaviour of three programming methods (“plain-C”, OpenMP and OpenCL) in DSP heterogeneous platforms is made. The results can be used in order to specify the most suitable programming method for a source-to-source code transformation tool transformation generating host and device code from the initial OpenMP/OpenCL single device code. The initial OpenMP/OpenCL single device code can be generated via a new backend for the “Bones” tool.

1.3. Organization

The remainder of the report is organized as follows. Chapter 2 surveys previous work in benchmarking DSP heterogeneous platforms, analytical performance models and source-to-source transformation tools. Chapter 3 presents the hardware that is going to be benchmarked within the project. The specification of each platform is given. The programming methods that are going to be examined during the porting procedure are given in Chapter 4. Their details and their differences are described. Chapter 5 describes the algorithms that are going to be used as an application for benchmarking purposes. The choice of the specific algorithms is clarified and an analysis of their characteristic is made. Chapter 6 presents the experiments that have been made during both the preparation and the final phase of the project. A set of guidelines and a methodology to distribute optimally an application over a heterogeneous platform are given in Chapter 7. Conclusions of the project are made in Chapter 8. An investigation regarding the feasibility of employing OpenMPI on RapidIO is made in Appendix A as future work. An introduction to Bones compiler is also made in Appendix A. Appendix B presents the K2H-PU benchmarking tool which was developed during the thesis project. Chapters 2 – 5 have already been presented in the preparation phase report.

2. Related work

This chapter introduces the related work that has already been conducted regarding the benchmarking of DSP heterogeneous platforms, the construction of analytical performance models for homogeneous and heterogeneous clusters and the development of source-to-source code transformation tools.

2.1. Benchmarking DSP heterogeneous platforms

According to [25] the traditional approaches (MIPS, MOPS, MACS) performance measurement could be misleading on processors. For example, the most common performance metric, MIPS (millions of instructions per second) is misleading since there could be great differences between processors' instruction sets. A more accurate performance metric is MOPS (millions of operations per second) but similarly there are differences in the definition of an operation between processors. In DSPs a common used metric is MACS (multiply-accumulates per second). Again, counting only the multiply-accumulates operations could be misleading. Moreover, the aforementioned metrics do not address secondary performance issues. Memory accesses and power consumption have to be considered for an efficient evaluation of a DSP heterogeneous platform. Finally, [26] states that if the DSP platform has not been designed for a specific application or if it is application-agnostic then the best option for the benchmarking procedure is to choose a computationally intensive algorithm. Several choices of this type of algorithms are given in [26]. In this work an image denoising algorithm (Chapter 5) has been selected.

Image denoising algorithms were selected as the use-case scenario given the fact that they can conform to the intrinsic target application's needs and provide incentives for research on an academic context. Moreover, they coordinate with future demands of the "Visual Solutions" sales activities spanning across multiple market segments (such as Industrial, Medical, Automotive, Defence & Aerospace, etc.) in which they can be deployed.

Parallel application performance is complex and often it is difficult to identify the potential bottlenecks and how future optimizations will impact performance. Performance models help to identify the bottleneck and can provide a good insight into how a compute system works. Moreover, performance models can guide developers into accurate optimization techniques [36].

In our investigation a newly introduced performance model is used. Roofline Model [11] ties together floating-point performance, operational intensity and memory performance in a 2D graph. The term "*operational intensity*" describes the number of operations per byte of memory traffic, defining total bytes accessed as those bytes that go to the memory after they have been filtered by the caches. In other words, operational intensity calculates the memory bandwidth needed by a kernel. Thus, there are three ways to improve performance according the Roofline model: increase computational performance, increase memory performance or increase operational intensity. Figure 2.1 illustrates how each factor affects Roofline model. According to Figure 2.1 and to the definition that is given above, in the Roofline model performance is the minimum of:

- Operation Intensity $*$ Bandwidth
- In-core Flop/sec
- In-core Flop/sec as a function of the floating point fraction

The multiple "*performance ceilings"* of the Roofline Model which are illustrated in Figure 2.1 present which optimizations can be implemented and in which way they improve the performance. For example in superscalar (e.g. x86) architectures by improving the instruction level parallelism (ILP) or by applying SIMD instructions the highest performance can be achieved. In this way computational bottlenecks can be reduced. Other optimization techniques can be used in order to reduce the memory bottlenecks. The performance of an application can be increased significantly by keeping many memory operations in flight. *Software prefetching* is usually used for that reason. Another alternative for extra memory bandwidth is the *restructure of loops for unit stride accesses*. In this way, hardware prefetching is engaged. A series of performance ceilings are introduced and discussed during the experimental analysis in Chapter 6.

A performance model like the Roofline model can provide realistic expectations of both performance and efficiency by presenting not only the hardware limitations for a given kernel but the potential benefit and priority of optimizations as well.

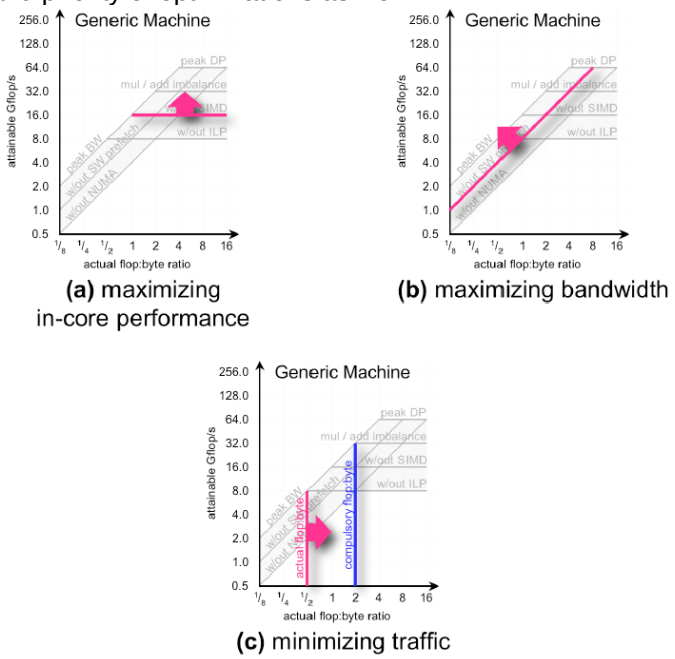

Figure 2.1 General ways to improve performance in Roofline model [11].

2.2. Analytical performance models

A more detailed description about the behaviour of a system can be given by an analytical performance model. Analytical performance models introduce an abstract description of a complicated system with mathematical equations by using the right assumptions and approximations. Therefore, a detailed understanding of the target hardware is needed. Analytical performance models are used in software performance testing since they are less time consuming than performance testing via detailed benchmarking.

Several analytical performance models have been proposed for both homogeneous [27] [28] and heterogeneous clusters [29] [30]. In both cases, symbolic expressions for different performance metrics have to be introduced in order to form the necessary mathematical functions which will be used to analyse the under examination system.

Depending on the number of the metrics that are used for the construction of the model, its complexity as well as its accuracy grows. The two metrics that form the key-parameters of every performance model are the *computational* and *communication cost*.

Regarding the heterogeneous clusters two main categories can be defined. In the first, each node of the cluster consists of cores of the same type while in the second category each node can contain a variety of cores. In both categories, the differences in the capabilities of each core have to be defined in order to make an accurate model. Another architectural feature which affects the construction of the performance model is the type of the communication between the nodes in the cluster. If a host node is used, then the individual factors of this node has also to be considered. Hence, both performance specifications and cluster architecture affect the accuracy and the predictability of an analytical performance model.

2.3. Source-to-source transformation

A lot of research has already been conducted on automating parallel programming (e.g. [20] [29] [31] [32]). [20] and [32] produce readable optimized parallel OpenMP/OpenCL/CUDA code. “Bones” [20] is based on skeleton-based compilation and “*algorithmic species*”, while [32] is based on the theory of convex array regions. Both approaches generate code for a single homogeneous (OpenMP) or heterogeneous (OpenCL/CUDA) multicore platform. On the other hand, [29] and [31] focus on automatic code parallelization for clusters. [29] proposes an automatic code generation tool for GPU heterogeneous clusters. The proposed tool gets as an input regular CUDA code for a single device and it outputs head (global host) and worker node code. In [31] a speculative DOALL (Spec-DOALL) parallelizing compiler for homogeneous clusters is implemented. The compiler takes sequential C/C++ source code as an input to generate parallelized code targeting the Cluster Spec-DOALL runtime.

3. Target Platforms

In this chapter the hardware that is selected as target platform to be benchmarked, is described. The specification of each platform is presented briefly.

3.1. PDAK2H

PDA2KH [4] is the platform that was employed for the thesis needs. Prodrive DSP AMC Keystone-II Hawking (PDAK2H) module is an Advanced Mezzanine Card which is assembled with a Keystone-II Hawking SoC from Texas Instruments. An additional Searay piggy back site module can be placed on the PDAK2H. More details about the specific platform follow.

3.1.1. PDAK2H technical characteristics

The PDAK2H is designed by Prodrive in a modular way so that it can be used in various systems. Typical deployment scenarios involve system purposes like video-processing, telecommunications or high-performance computer processing.

The AMC modules are placed in an AdvancedTCA (ATCA) shelf. *Figure 3.1* shows a diagram of the ATCA shelf. The PDAK2H can be placed in any AMC bay. PDAK2H modules can communicate via SRIO and Ethernet.

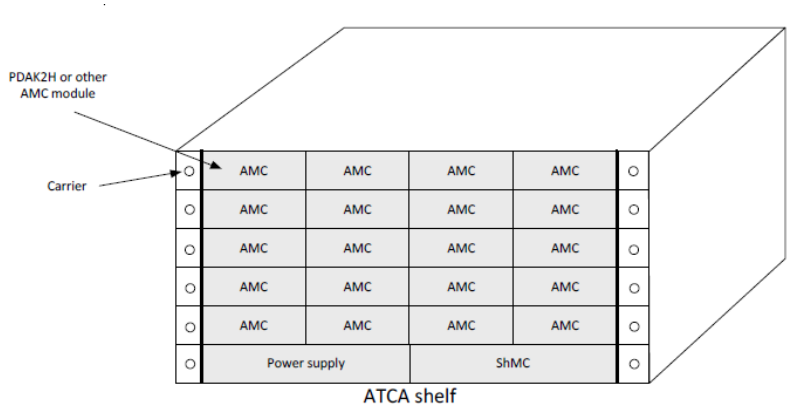

Figure 3.1 ATCA shelf [5]

The features and the blocking diagram of PDAK2H follow.

PDAK2H Features:

- System on Chip
  - TI 66AK2H14 (Keystone-II/Hawking)
  - 8x C66x CorePac DSP
  - 4x ARM Cortex A15 MPCore
- DDR3 SDRAM
  - Up to 8 GB for DSP
  - Up to 2GB for ARM
  - Up to 667 MHz (1333 MT/s data rate)
- Searay piggy back site
  - 25+ Gbps
  - 2x 1000 BASE-BX Ethernet differential interface
  - 1x SRIO quad lane differential interface
  - 2x Hyperlink quad lane differential interface
  - Hyperlink up to 6.25 Gbaud Operation per lane

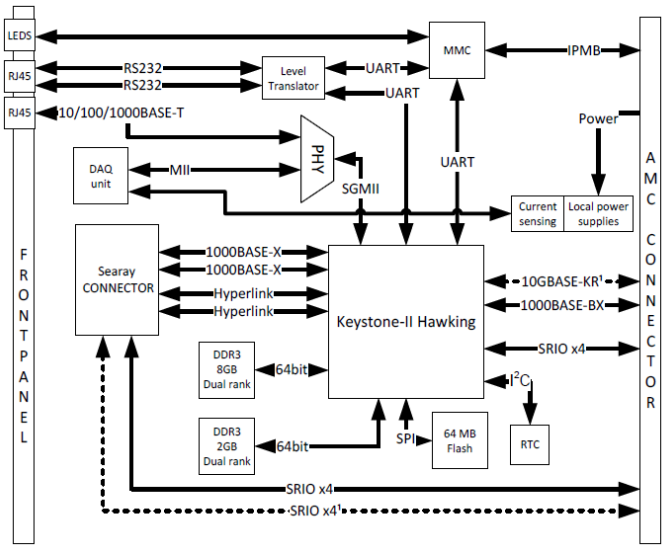

Figure 3.2 PDAK2H Block Diagram [5]

Moreover, regarding the additional Searay piggy back site module that can be placed on the PDAK2H, during experimentation the APBC66: AMC Piggy Back C66x (Shannon) module [5] has been selected to be the extension module for the PDAK2H. Figure 3.3 shows APBC66.

Figure 3.3 APBC66 Piggy Back

The features and the block diagram of APBC66 module follow.

APBC66 features:

- 2x System on Chip
  - TI C6678 SoC
  - 8x C66x DSP Core
  - Up to 1.2GHz operation frequency
- DDR3 SDRAM
  - Up to 8GB per SoC
  - Up to 667MHz, (1333MT/s)

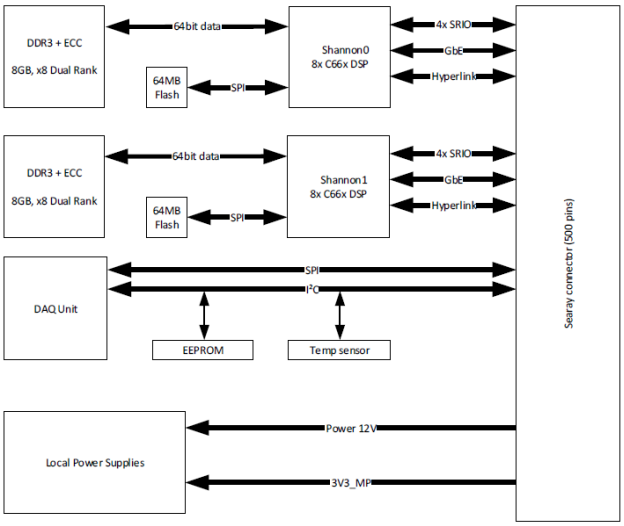

Figure 3.4 APBC66 Block Diagram [6]

Some of the most important features of the C66x DSP core, with which both TI 66AK2H14 and TI C6678 SoC are equipped, are listed below:

- Integrated fixed- and floating-point operation
- 8-way VLIW
- SIMD operations for fixed point
- Fully-pipelined instructions
- Up to eight, 32-bit multiplies per cycle
- Up to two double-precision multiplies per cycle

3.1.2. PDAK2H usage during experimentation

Different configurations of the available processors are going to be benchmarked. By varying the number of involved ARMs/DSPs, it will be able to investigate the scalability issues of PDAK2H. Moreover, by using different porting methods it will be able to find advantages and disadvantages of each one. Figure 3.5 shows the block diagram of PDAK2H with the additional APBC66 module.

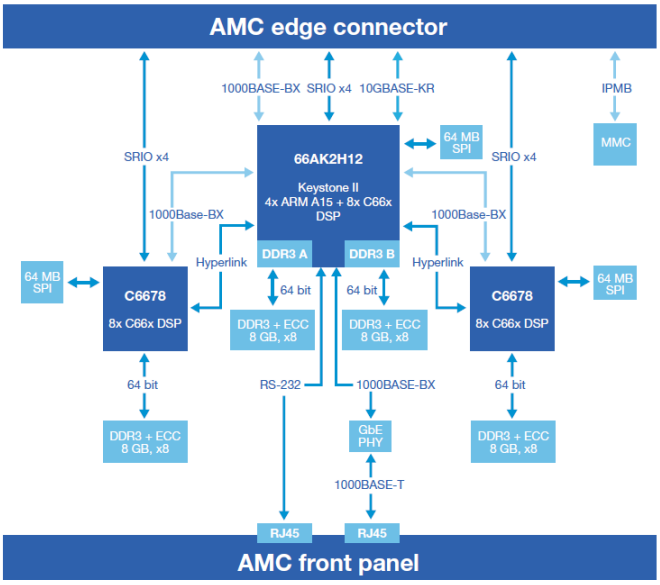

Figure 3.5 PDAK2H with APBC66 Block Diagram

3.2. ATCA-TK2-6PU Blade

The thesis reserach also focuses in the benchmarking of multiple identical PU's. For this scope, the Prodrive ATCA-TK2-6PU Blade (Figure 3.6) has been selected. ATCA-TK2-6PU is an ATCA blade which is assembled with multiple PUs. Each PU contains: one TI Keystone-II (66AK2H14) and two Keystone-I (TMS320C6678) SoCs (see also Figure 3.7). Multiple ATCA-TK2-6PUs can be used together in an ATCA shelf and communicate via 10Gb Ethernet.

The features and the block diagram of the ATCA-TK2-6PU follows.
ATCA-TK2-6PU features:

- 6x Processing Unit (PU), each containing
  - 1x TI 66AK2H14 (Keystone-II / Hawking)
    - 8x C66x CorePac DSP (@1.2GHz)
    - 4x ARM Cortex A15 MPCore (@1.4GHz)
    - 8 GB DDR3 SDRAM on controller A
      - Up to 800 MHz (1600 MT/s data rate)
    - SRIO gen 2 (x4 Gen 2) to zone 3
    - Hyperlink interface (x4 10G)
  - 2x TI TMS320C6678 (Keystone-I / Shannon) (@ 1.2 GHz)
    - 8x C66x CorePac DSP
    - 8 GB DDR3 SDRAM
      - Up to 667 MHz (1333 MT/s data rate)
    - Internally interconnected to 66AK2H14 by Hyperlink (x4 10G)
    - SRIO (x4 Gen 2)
- Ethernet interface
  - Onboard 10GbE switch
  - 6x 10GBASE-KR connected to 6 PUs
- SRIO switch
  - 2x onboard SRIO switch (IDT CPS1848)
  - 4x SRIO (x4) gen 2 to zone 3
  - 3x SRIO (x4) gen 2 to each PU

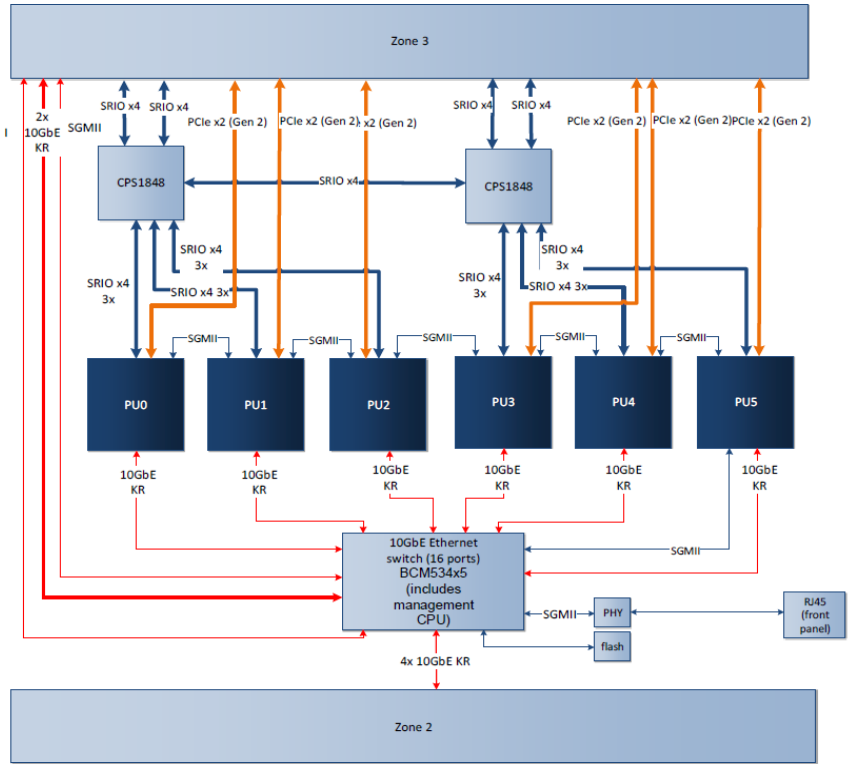

Figure 3.6 ATCA-TK2-6PU Block Diagram [6].

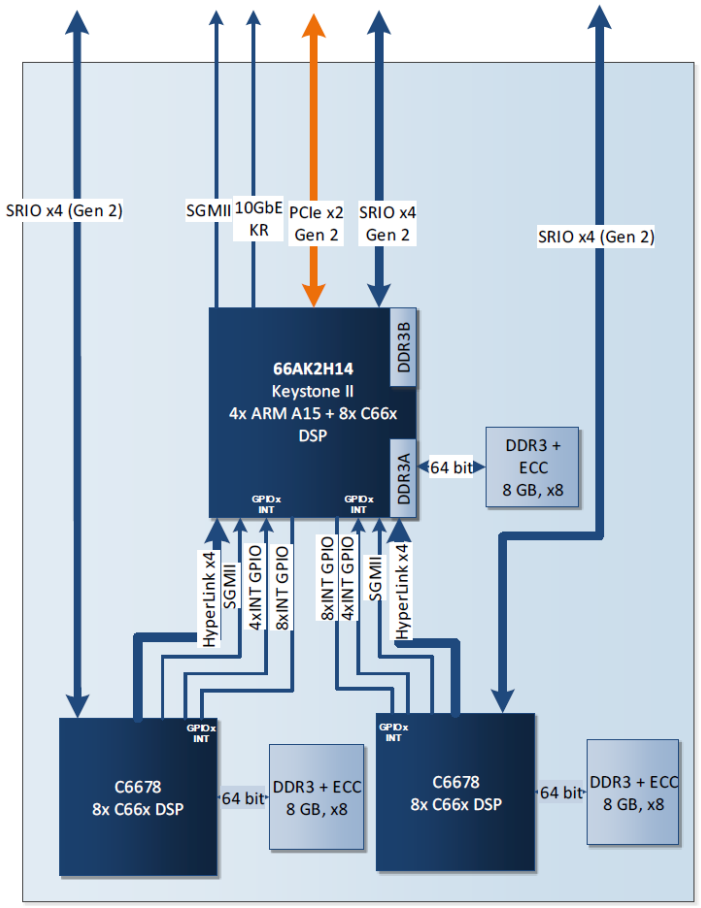

Figure 3.7 One PU block diagram [6].

By varying the number of involved PUs or by using multiple blades, multiple different configurations of the ATCA-TK2-6PU can be created. A detailed benchmark analysis will follow in order to show not only the capabilities but also the scalability issues of the platform.

4. Parallel Programming Methods

In this chapter, three programming methods for programming the CPU as well as the DSP are introduced. Their details as well as their differences are described.

In parallel programming, part of the program is divided into smaller parts in order to be distributed to multiple processing units. The smaller parts of the program will be processed by the processing units simultaneously (in parallel).

The programming methods which are used to divide and distribute the parallelizable part can be valued by their generality (how well different problems can be expressed for a variety of different architectures) and their performance (how efficiently they divide and distribute the parallelizable part of the program).

4.1. “Plain-C”

In the rest of the document, the manual parallelization of an algorithm and its distribution over a multicore homogeneous or heterogeneous platform will be termed as the “*plain-C*” porting method.

4.1.1. CPU side

The individual parts of a parallelized algorithm can be offloaded among several CPUs via the `pthread` or the OpenMP library. The following code snippets illustrate an example where the algorithm has been parallelized into four parts in order to be distributed to four available cores. First the pthread implementation is presented and then the OpenMP follows.

```
void *nlm(void *arguments)
{
    if (arguments) {
        struct thread_struct *args = arguments;
        int rows_per_arm = SIZE / 4;
        int start_row = (args->id)*rows_per_arm;
        int end_row = start_row+rows_per_arm;
        for(int py=start_row; py<end_row; py++) {
            kernel();
        }
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{


    for(t = 0; t < 4; t++) {
        args.id = t;
        rc = pthread_create(&threads[t], NULL, nlm, (void *)&args[t]);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
```

```
omp_set_num_threads(4);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int rows_per_arm = SIZE / 4;
    int start_row = (id)*rows_per_arm;
    int end_row = start_row+rows_per_arm;
    for(int py=start_row; py<end_row; py++) {
        kernel();
    }
}
```

4.1.2. DSP side

Regarding the DSP side of a platform, the parts of a parallelized algorithm can be offloaded with the help of the MPM TI libraries. The host program which includes the necessary libraries runs in the CPU side and loads the binary files in the DSP side. The following sample program presents an example of the above method. Firstly, all the available DSPs are reset. The binary files are loaded in each DSP and finally DSPs executes the loaded application.

```
// reset the available DSPs
for (int i=0;i<NR_DSPS;i++){
     mpm_reset(dsp[i],err);
}
// load the individual parts of code to the available DSPs
for (int i=0;i<NR_DSPS;i++){
    mpm_load(dsp[i],part_of_code[i],err);
}
// run the loaded parts of code on the available DSPs
for (int i=0;i<NR_DSPS;i++){
    mpm_run(dsp[i],err);
}
```

4.1.3. Heterogeneous platform

A combination of the above two methods can be used in order to use both CPU and DSP side. In this way the capabilities of both sides can be used.

4.2. OpenMP

Open Multi-Processing (OpenMP) [1], is an API for parallel programming in C, C++ and Fortran that is portable across shared memory architectures from different vendors. The directives of OpenMP API extend the mentioned programming languages, and by using the provided library routines and environment variables it is able to control the runtime behaviour of the program. The main advantage of OpenMP is its simple and flexible interface.

4.2.1. Execution Model

The OpenMP API is a multithreading paradigm and it uses the fork-join model of parallel execution. More precisely, a master thread forks a specified number of slave threads in order to execute a task in parallel. All the necessary resources are allocated to these threads. Figure 4.1 demonstrates the above.

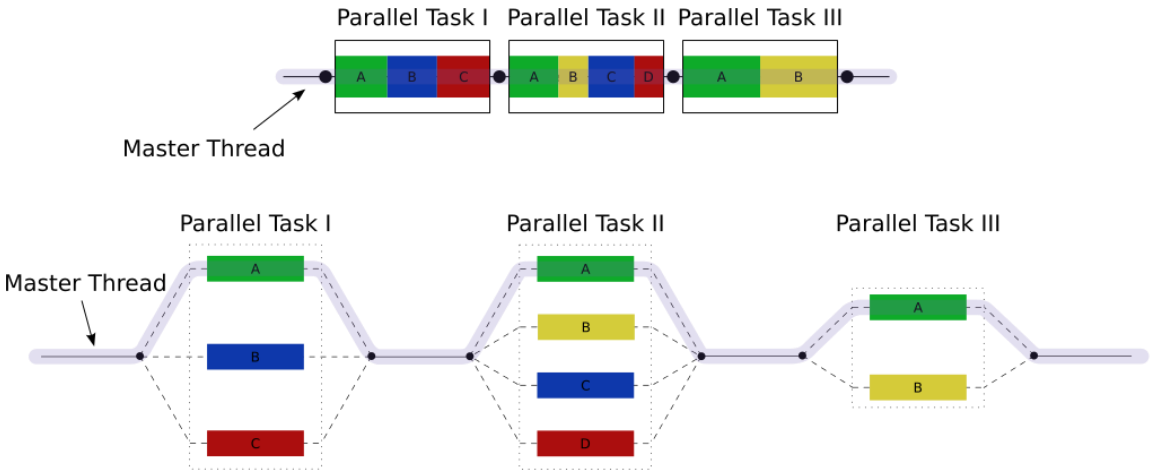

Figure 4.1 Master/Slave Multi-Threading programming (in top: the serial execution is presented, in bottom: the parallelization of each task in multiple threads is presented) [1]

Developers are responsible for the correct execution of the programs after the introduction of the OpenMP directives. Dependencies as well as parallelizable parts of the programs must be discovered by the developer in order to add the necessary OpenMP directives wherever are applicable. In this way, OpenMP enhances the performance of the program without altering the results by supporting both data- and task-based parallel programming models.

4.2.2. Memory Model

Regarding the memory model, the OpenMP API provides a relaxed-consistency, shared-memory model. Reads and writes are allowed to be completed out of order, but synchronization operations used to enforce ordering. The relaxed models are divided based on what read and write orderings they relax [2]. Each thread, can have its own temporary view of the shared memory which is used by all threads to store and retrieve variables. However, temporary view of memory is not always consistent with memory. Though, the OpenMP flush operation enforces consistency between the temporary view and memory. Finally, threads have private memory for private data. Each thread has a stack for data local to each task it executes.

4.2.3. Parallel Regions

Using “omp parallel” pragma, the programmer creates threads. For example, the following code excerpt presents an example where a 10-Thread Parallel region is created. Each thread redundantly executes the code in the structured block. The `do_it_in_parallel(id,A)` is called for id = 0 to 9.

```
float A[10000];
omp_set_num_threads(10);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    do_it_in_parallel(id,A);
}
```

4.2.4. OpenMP in a heterogeneous platform

OpenMP can also be used in heterogeneous platforms. OpenMP 4.0 supports accelerators/coprocessors. One host device is able to offload target regions to target devices. The target devices are multiple accelerators/coprocessors of the same kind. The above is done using the *target* construct. With the specific construct it is able to map variables to a device data environment and to transfer control from the host to the device. The following code snippet shows an example of target construct where arrays *a*, *b* and variable *N* are copied from host memory to target (coprocessor) memory and array c is copied from target memory to host memory. The iterations of the for-loop are going to be divided to all the available coprocessors, while the host device will wait until the target region is completed.

```
#pragma omp target map(to: N, a[0:N], b[0:N]) map(from: c[0:N])
{
    int i;
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

4.3. OpenCL

Open Computing Language (OpenCL) [3] is an open standard for general purpose parallel programming across heterogeneous platforms consisting of CPUs, GPUs, DSPs and other processors.

4.3.1. Platform Model

Figure 4.2 illustrates the OpenCL platform model where the host is connected to multiple OpenCL devices. Moreover, OpenCL devices are divided into one or more compute units which are also divided into one or more processing elements. The processing elements are responsible for the computations inside a device. The host provides OpenCL APIs while the Processing Elements execute OpenCL kernels written in a variant of C99. The specific model is very similar to the model that OpenMP 4.0 provides for heterogeneous platforms.

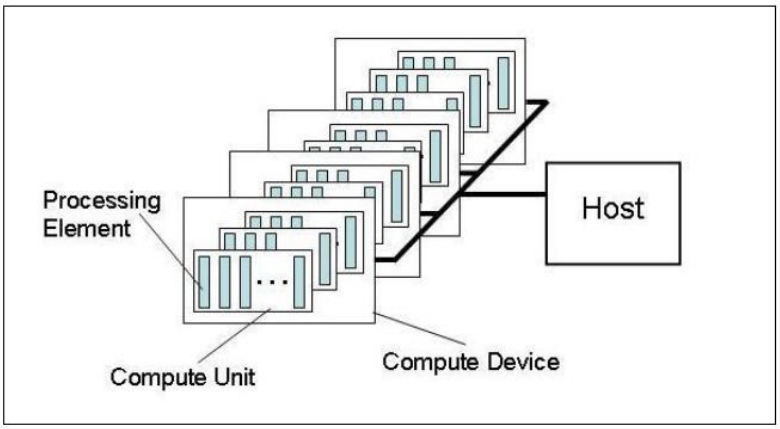

Figure 4.2 OpenCL platform model [3]

Two types of platform profiles are available. In the first type, *full profile*, an online compiler is available for all the devices. In this case, the kernel (the code that is executed in the compute device, Figure 4.2) is built from source during runtime using the OpenCL runtime library. Hence, it provides a more flexible and adaptive compiling which is not dependent on the hardware. On the other hand, *embedded profile,* which is a reduced-functionality profile, an online compiler is not required. Offline compilation is used in this profile. Since the kernel is pre-built, the time interval between starting the host code and the kernel getting executed is negligible. The disadvantage of this method is that it is more platform-dependent, than online compilation.

4.3.2. Execution model

Three types of commands are used by the host in order to control a device, namely:

1. Kernel-enqueue commands: Enqueue a kernel for execution on a device
2. Memory commands: Memory-related commands (data transfer etc.)
3. Synchronization commands: Order constraints between commands

All the commands, which are provided by the host, are placed in a command-queue which is distinct for each device. Furthermore, commands that belong in the same command-enqueue are able to be executed in either: in-order or out-of-order. In the first mode, commands are executed in the same order they are enqueued to their command-queue. On the other hand, in out-of-order execution mode, commands follow only synchronization rules or restrictions related to dependencies.

Each kernel execution defines a *kernel-instance*, which includes a number of functions called *work-items*. The work-items are managed by *work-groups*, which are parts of the kernel's *index space*. Figure 4.3 shows an example of an *NDRange* index space and the mapping of the work-items onto work-groups of $(S_x * S_y)$ size. The "ready to execute" work-groups are placed into a *work-pool* where they wait for their execution on the compute units of the device. Once the work-groups are inside the work-pool, they can execute in any order. Hence, there is no safe and portable way to synchronize across the independent execution of work-groups. However, there are high-level synchronization constructs that apply to all the work-items in a work-group.

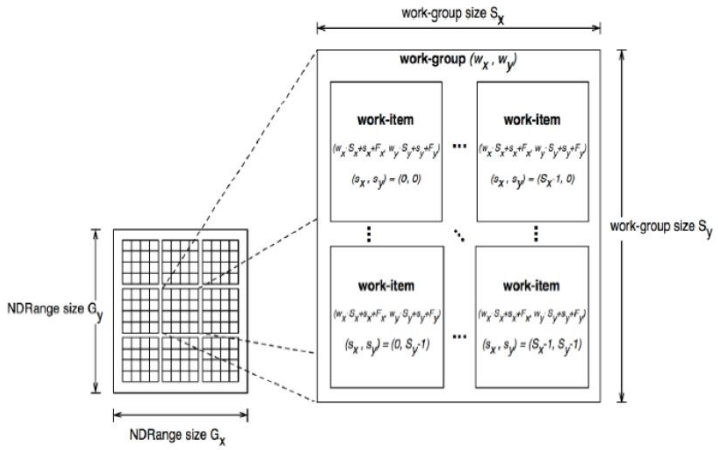

Figure 4.3 Index space - work-group - work-item [3]

4.3.3. Memory Model

The memory model in OpenCL is defined in four levels:

1. Global memory: shared by all processing elements (compute devices and host device)
2. Constant memory: writable only by the host device
3. Local memory: shared by a group of processing elements
4. Work item private memory

Figure 4.4 illustrates the memory model provided by OpenCL. It is a relaxed-consistency memory model where consistency is enforced by explicit synchronization constructs. Moreover, it is possible for a device to exclude some levels of memory hierarchy.

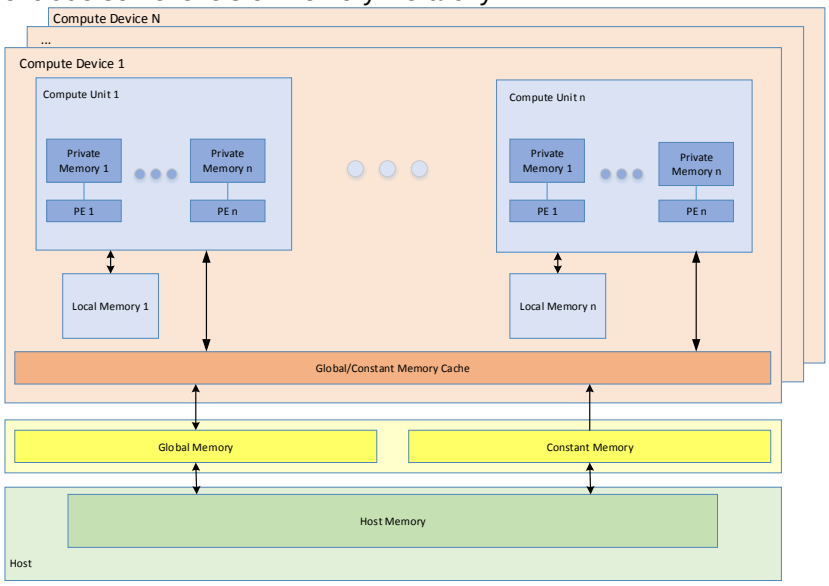

Figure 4.4 OpenCL Memory Model

4.3.4. OpenCL implementation

As it has already been mentioned OpenCL code is divided into host and device code.

4.3.4.1. Host Code

First, the necessary amount of memory has to be allocated, by a memory reservation call, in order to verify that there is available memory and to set this amount of memory inaccessible by other activities. The creation of a command queue follows and the OpenCL kernel is ready to be loaded by the `LoadOpenCLKernel` function. The three following functions are responsible for the transformation of the kernel into an OpenCL program description (`clCreateProgramWithSource`), for its compilation (`clBuildProgram`) and the creation of the

kernel object (*clCreateKernel*). Functions regarding the memory allocation in the compute device follow. Finally, the *clEnqueueNDRangeKernel* function enqueues a command to execute a kernel on a device and when the results are ready, are read via the *clEnqueueReadBuffer* function. The following code excerpt presents a simplified example of OpenCL host code.

```
/*
    Host Memory Allocation
                                */

// Connect to a compute device
   int gpu = 1;
   err = clGetDeviceIDs(platform_ids[0], gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU, 1,
   &device_id, NULL);

// Create a compute context
   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// Create a commands queue
   commands = clCreateCommandQueue(context, device_id, 0, &err);

// Create the compute program from the source file
   char *KernelSource;
   long lFileSize;

   lFileSize = LoadOpenCLKernel("matrixmul_kernel.cl", &KernelSource, false);
   program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL,
&err);

// Build the program executable
   err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create the compute kernel in the program we wish to run
   kernel = clCreateKernel(program, "matrixMul", &err);

/*
    Device Memory Allocation
                                */

// Enqueue a command to execute a kernel on a device
   err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, globalWorkSize, localWorkSize,
0, NULL, NULL);

// Read the result memory on the device and copy it to the host memory
   err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, mem_size_C, h_C, 0, NULL, NULL);
```

4.3.4.2. Device Code

The following sample program presents an example of OpenCL device (GPU) kernel code. The *__global* identification describes variables which are located in the external memory (*__local* identification is used for those in the shared memory). The `get_global_id` function determines the thread id and in the above example is used to specify the matrix location.

```
/*
kernel.cl
* Matrix multiplication: C = A * B.
* Device code.
*/

// OpenCL Kernel
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB){

    int tx = get_global_id(0);
    int ty = get_global_id(1);

    // value stores the element that is
    // computed by the thread
    float value = 0;
    for (int k = 0; k < wA; ++k){
        float elementA = A[ty * wA + k];
        float elementB = B[k * wB + tx];
        value += elementA * elementB;
    }

    // Write the matrix to device memory each
    // thread writes one element
    C[ty * wA + tx] = value;
}
```

5. Selected Algorithms

The algorithms that were selected to be investigated, are described. In order to test the capabilities of the platforms the following two compute-intensive image denoising algorithms are considered:

- Non-Local Means (NL-M)
- Block Matching 3D (BM3D)

As described in 5.4, NL-M is selected as the most suitable benchmark.

5.1. Image denoising algorithms

From the moment that an image is captured a variety of natural (i.e. photon counting process) sources can introduce noise and decrease the quality of the image. Moreover, additional image processing (contrast enhancement, blur removal, etc.) can amplify noise which has to be reduced for a variety of reasons such as aesthetic purposes in marketing or for practical purposes such as computer vision.

First denoising algorithms such as Gaussian smoothing model [7] or the anisotropic filtering model [8] were used to eliminate the noise by replacing the color of a pixel with the weighted average of the colors of the nearby pixels. In linear smoothing filters the noise elimination is attempted by convolving the original noisy image with a low-pass filter. In this way, the original pixels are replaced by the average value, or a weighted average, of themselves and their close neighbors. In the Gaussian smoothing model, the set of weights is determined by the Gaussian filter. In anisotropic filtering model, the denoising procedure is based on a smoothing partial differential equation which is called anisotropic diffusion. Again, Gaussian filtering can be used but with a diffusion coefficient designed to detect edges in order not to blur the edges of the image.

5.2. Non-Local Means (NL-M)

The main characteristic of the above two examples is the fact that they take into account only the spatial difference between two pixels. On the other hand, a third example of image denoising algorithms, Non-Local Means algorithm [9] is more data dependent as it takes into account both spatial and photometric distances. A second difference, which actually introduces the high compute-intensity of the algorithm, is that it searches for similar pixels by comparing a whole window around each pixel (patch). Hence the point-wise measurement which is performed in the previous two algorithms is enhanced and replaced by a patch-wise one. The denoising is then done by computing the weighted average color of the most similar pixels. For an area Ω of an image, discrete NL-M algorithm is defined by the formula

$$u(p) = \frac{1}{C(p)} \sum_{q \in \Omega} u(q) f(p,q)$$

Where, $u(p)$ is the filtered value of the image at pixel p, $u(q)$ is the unfiltered value of the image at pixel q, $f(p,q)$ is the weighting function which determines how similar pixel p is to pixel q and $C(p)$ is a normalizing factor, given by:

$$C(p) = \sum_{q \in \Omega} f(p,q)$$

A common choice for the weighting function is the Gaussian function which is given by:

$$f(p,q) = e^{-\frac{|B(q)-B(p)|^2}{h^2}}$$

Where, h is the filtering parameter and $B(p) = \frac{1}{|R(p)|} \sum_{i \in R(p)} u(i)$ is the local mean value of the pixels which surround p and they are included in the square region, called patch, $R(p)$. A graphical representation of NL-M is given in Figure 5.1.

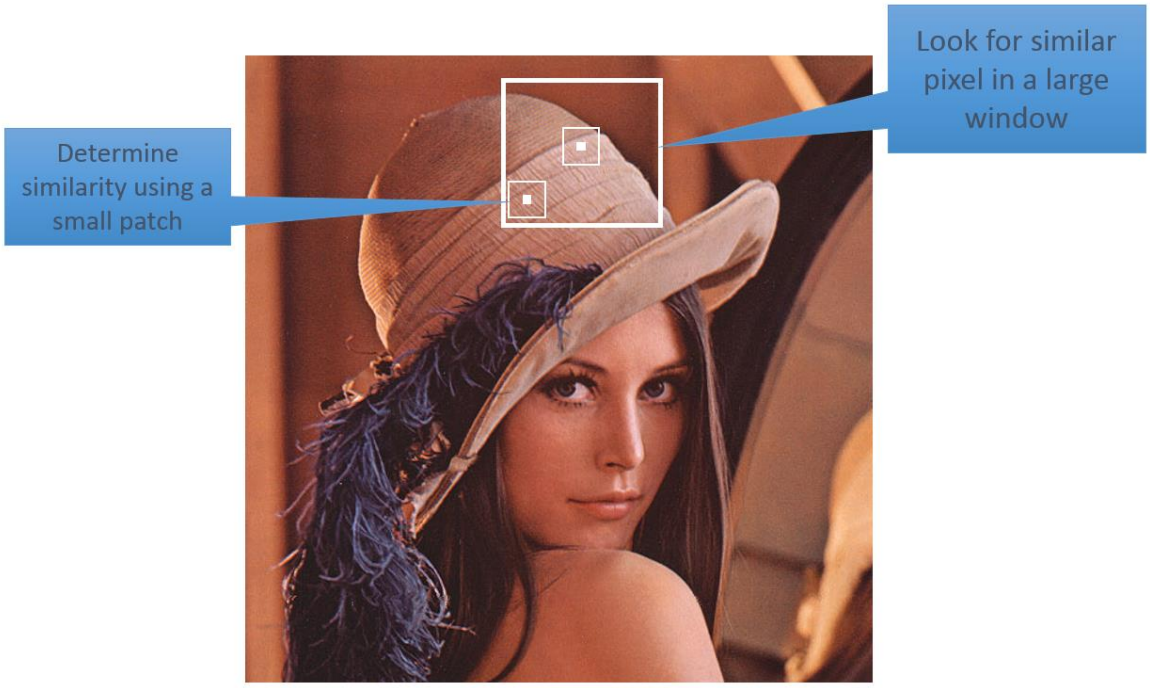

Figure 5.1 NL-M graphical representation

The following code snippet shows an implementation of NL-M for a 21x21 window and a 3x3 patch on a 512x512 image. Three floating point operations are required for each patch and extra six in each window. Using the following formula, the number of FLOPs which are needed for the denoising of a single pixel can be computed, for a window and a patch of sizes $(2 * R + 1) * (2 * R + 1)$ and $(2 * F + 1) * (2 * F + 1)$, respectively:

$$FLOPS/Pixel = (2 * R + 1)^2 * (2 * F + 1)^2 * 3 + (2 * R + 1)^2 * 6$$

Hence, for the specific sizes of the window and the patch approximately 15k FLOPS are needed for the denoising of a single pixel. The above can give a first estimate for the excessive compute intensity of the algorithm. However, due to the structure of NL-M its parallelization is very easy. The original image can easily be divided into equal parts and shared to all the available processing units since there are no data dependencies between windows.

```
for (int py=11; py<501; py++) {
    for (int px=11; px<501; px++) {
        float Cp = 0, sum = 0;
        for (int qy=-10; qy<=10; qy++) {
            for (int qx=-10; qx<=10; qx++) {
                float d = 0;
                for (int fy=-1; fy<=1; fy++) {
                    for (int fx=-1; fx<=1; fx++) {
                        float pix_p = in[py + fy][px + fx];
                        float pix_q = in[py + qy + fy][px + qx + fx];
                        float delta = pix_p - pix_q;
                        d += delta * delta;
                    }
                }
                float dd = (1.0f / 9.0f) * d;
                float w = expf(-1.0f*fmax(dd-2.0f*S*S, 0.0f)/(H*H));
                Cp += w;
                sum += in[py + qy][px + qx] * w;
            }
        }
        out[py][px] = (1.0f/Cp) * sum;
    }
}
```

5.3. Block Matching 3D (BM3D)

A similar, but much more compute intensive, to NL-M denoising algorithm is Block Matching 3D (BM3D) [10]. Similar to NL-M, BM3D is data dependent and it uses patch-wise measurements. However, BM3D introduces a 3D transform domain which helps a lot in the quality of the output denoised image. Moreover, is composed by two major steps of:

1. Basic estimation using hard threshlolding during the filtering
2. Wiener filtering using the combination of the original image and the basic estimation from the first step

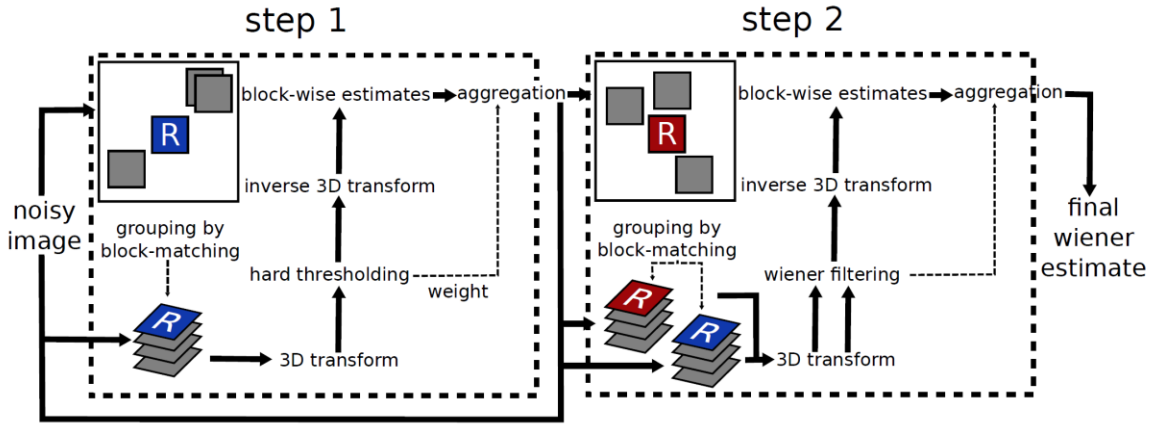

Figure 5.2 Scheme of BM3D [10]

As Figure 5.2 illustrates the input of the first step is the original noisy image. Reference patches *R* (windows around each pixel) are going to be created. Then, the noisy image is going to be searched in R-centered windows (bigger windows around the patch *R*) for patches similar to R. The 3D group is then built by stacking up the similar patches. The collaborative filtering follows, where a 3D isometric linear transform is applied to the 3D group, followed by a shrinkage of the transform spectrum via hard threshlolding. The value of the threshold is determined by the user. An inverse 3D transformation is performed which has as a result a single estimate for each used patch and a variable number of estimates for every pixel. Hence, an averaging procedure of those estimates, called aggregation, follows which is described by the following formulas. $u^{basic}(x)$ is the output of the first step.

$$u^{basic}(x) = \frac{\sum_P w_P^{hard} \sum_{Q \in \mathcal{P}(P)} \mathcal{X}_Q(x) u_{Q,P}^{hard}(x)}{\sum_P w_P^{hard} \sum_{Q \in \mathcal{P}(P)} \mathcal{X}_Q(x)}$$

Where, $w_P^{hard} = \begin{cases} (N_P^{hard})^{-1}, if\ N_P^{hard} \geq 1 \\ 1, otherwise \end{cases}$ and $\mathcal{X}_Q(x) = \begin{cases} 1, x \in Q \\ 0, otherwise \end{cases}$

N_P^{hard}: Number of non-zero coefficients in the 3D block after hard-threshlolding
$u_{Q,P}^{hard}(x)$: The estimate of the value of the pixel x belonging to the patch Q obtained during collaborative filtering of the reference patch P

Regarding the second step of BM3D, it receives as an input both the noisy image and the basic estimation of the first step. It continues by patch-matching only on the basic estimate and it proceeds with two 3D transformations, by stacking up the patches from u^{basic} and by stacking up, in the same order, patches from the noisy image. The collaborative filtering follows using Wiener coefficients. After an inverse 3D transformation, the aggregation procedure follows. Its formulas are presented below.

$$u^{final}(x) = \frac{\sum_P w_P^{wien} \sum_{Q \in \mathcal{P}(P)} \mathcal{X}_Q(x) u_{Q,P}^{wien}(x)}{\sum_P w_P^{wien} \sum_{Q \in \mathcal{P}(P)} \mathcal{X}_Q(x)}$$

Where, $w_P^{wien} = \|w_p\|_2^{-2}$ and $\mathcal{X}_Q(x) = \begin{cases} 1, x \in Q \\ 0, otherwise \end{cases}$

$u_{Q,P}^{wien}(x)$: The estimate of the value of the pixel x belonging to the patch Q obtained during collaborative filtering of the reference patch P

The above analysis of BM3D concerns grey level images. The adaptation of the algorithm to color images can be done by the following procedure:

1. Transformation of the RGB noisy image to a luminance-chrominance space (YUV). Y denotes the luminance channel, and U, V the chrominance channels.
2. For both major steps of BM3D:
   - Grouping is only performed with the Y channel
   - The 3D block built on Y is used for all three channels
   - Collaborative filtering is applied to each channel separately as well as the aggregation procedure
3. Inverse transformation to return to the RGB space

Depending on the choice of the color space transform, important variations are presented in the quality of the image according to [10]. YUV transform has been selected to be described because of its simplicity.

In addition to the extra second step of BM3D which introduces further compute-intensity, BM3D introduces extra compute-intensity because of its 3D filtering. NL-M performs only a patch average which corresponds to a 1D filter in the 3D block. On the other hand, BM3D proceeds in a 3D filtering of the three dimensions of the computed blocks which leads to a greater quality of the output image and to a much higher compute-intensity.

Similar to NL-M, BM3D can be parallelized by dividing the original image into multiple parts. Moreover, a task parallelization is also feasible by dividing the two major steps to different processing units.

5.4. NL-M, BM3D comparison

During the preparation phase it became clear that the specific implementation of NL-M cannot be used in real-time applications when the PDAK2H is used as the target platform.

Based on experiments that have been conducted, BM3D in general was more than two times slower than NL-M. This can be explained by its complexity which has been already been discussed. Moreover, according to [10], the quality of the output in terms of PSNR is close to the PSNR for NL-M. Figure 5.3 illustrates the comparison between NL-M and BM3D and indicates that significant difference between their complexity and the minor difference in the quality of the output image.

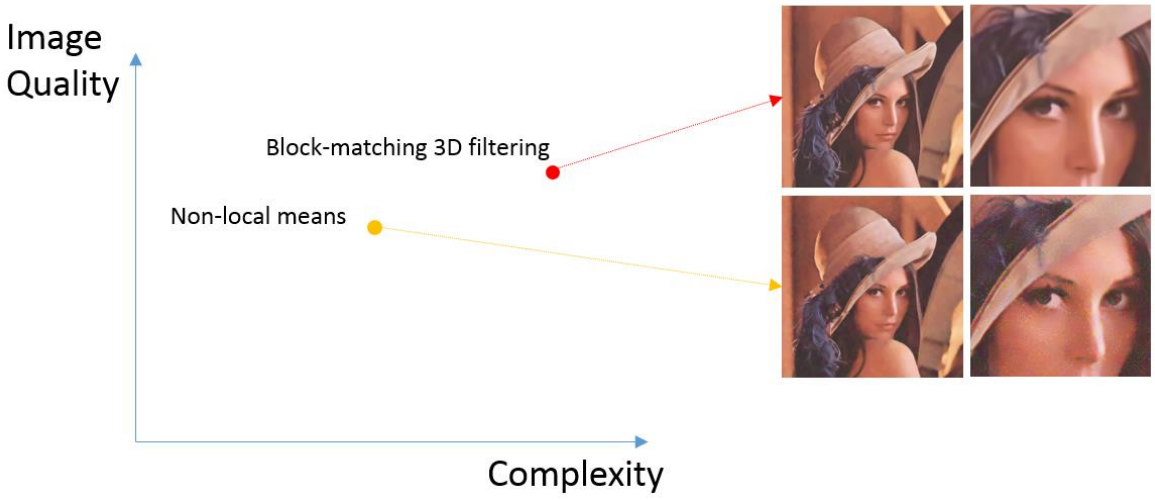

Figure 5.3 NL-M, BM3D comparison

Due to the above and due to the fact that the thesis objective is benchmarking via the use of an image denoising algorithm which can be used in a real-time application, it was decided to continue the research by taking into account and porting only NL-M and not BM3D in the already described platforms.

6. Experimentation

In this chapter the experiments which were performed on PDAK2H and ATCA-TK-ATCA 6PU blade are discussed.

6.1. Experimental setup

This section describes the experimental setup used during the benchmarking procedure of this thesis. The experimental setup is comprised from the target hardware and the software application that was developed for the benchmarking of the hardware.

6.1.1. Hardware

Two proprietary heterogeneous multi-SoC platforms were used for the thesis objectives validation:

1. PDAK2H (Figure 6.21) and
2. ATCA-TK2-6PU Blade (Figure 6.25)

The main component of both platforms is the TI Keystone-II “Hawking” SoC (Figure 6.15).

Details about the target hardware are given in Chapter 3.

6.1.2. K2H-PU Benchmarking tool

For the experimentation on the aforementioned configurations, the K2H-PU Benchmarking tool has been developed. The K2H-PU benchmarking tool has as a goal to benchmark and demonstrate the capabilities of processing units whose main component is the TI Keystone-II “Hawking” SoC. Two examples of such processing units are the PDAK2H and the ATCA-TK2-6PU blade.

The user is able to give as input a compute-intensive algorithm and the input-data that are going to be processed. The programming method (“plain-C”, openMP, openCL described in Chapter 4) and the K2H-PU configuration (PDAK2H or ATCA-TK2-6PU with the specified number of ARMs, DSPs, and PUs) are the parameters that can be configured by the user. After the execution of the algorithm, the output-data and the aggregated results are given by the tool as output.

Finally, the user is able to use the tool both remotely from a host Windows/Linux machine and locally from a processing unit. Figure 6.1 presents a test-case of the Graphical User Interface (GUI) of the K2H-PU benchmarking tool where NL-M is selected as the benchmarking application, the “plain-C” as the programming method and the 8 DSPs of the TI Keystone-II “Hawking” SoC of the PDAK2H as the target platform.

For more details about the functional design and the operating principle of the tool see Appendix B.

Figure 6.1 K2H-PU benchmarking tool

6.2. Experimentation scope

In this section the experimental objectives that were set during this thesis and the experimental procedure that was followed are described.

6.2.1. Experimental objectives

Three main experimental objectives were set during the research attempt that was made:

- "Plain-C" experimentation
- Programming method comparison
- Analytical performance models

6.2.1.1. "Plain-C" experimentation

During the preparation phase of the graduation assignment the TI Keystone-II "Hawking" SoC of PDAK2H has been benchmarked. Experiments were performed on both types of available processors (ARM A15 and DSP C66x) while their scalability issues were investigated. The roofline model was constructed and analyzed for both ARM and DSP cores and a series of performance ceilings are discussed. Heterogeneous configurations were examined by combining both types of processors. Section 6.3 presents the results from the preparation phase and describes the further experimentation that was made.

6.2.1.2. Programming method comparison

NL-M was ported onto the TI Keystone-II "Hawking" SoC with the help of three programming methods:

1. "plain-C"
2. OpenMP
3. OpenCL

The differences of the three programming methods were investigated while they were compared based on both the achieved performance and their usability.

Section 6.4 presents the results from programming method comparison.

6.2.1.3. Analytical Performance Models

Analytical performance models are developed and verified for:

1. TI-Keystone-II Hawking SoC
2. PDAK2H
3. ATCA-TK2-6PU Blade

A theoretical analysis was firstly conducted and the necessary experiments for its validation were followed.

Section 6.5 presents the analytical performance models and their verification.

6.2.2. Experimental procedure

Figure 6.2 illustrates the steps and paths that were addressed during the experimental procedure of the complete graduation assignment.

- Two heterogeneous platforms (Chapter 3) were benchmarked.
- By varying the number of available processing units in each platform, multiple configurations can be created (TI Keystone-II "Hawking" Example: [Table 6.1]).
- Based on the comparison of the selected algorithms (Chapter 5) NL-M has been selected to be ported in the aforementioned configurations.
- Three different programming methods (Chapter 4) were used and examined during the porting procedure.

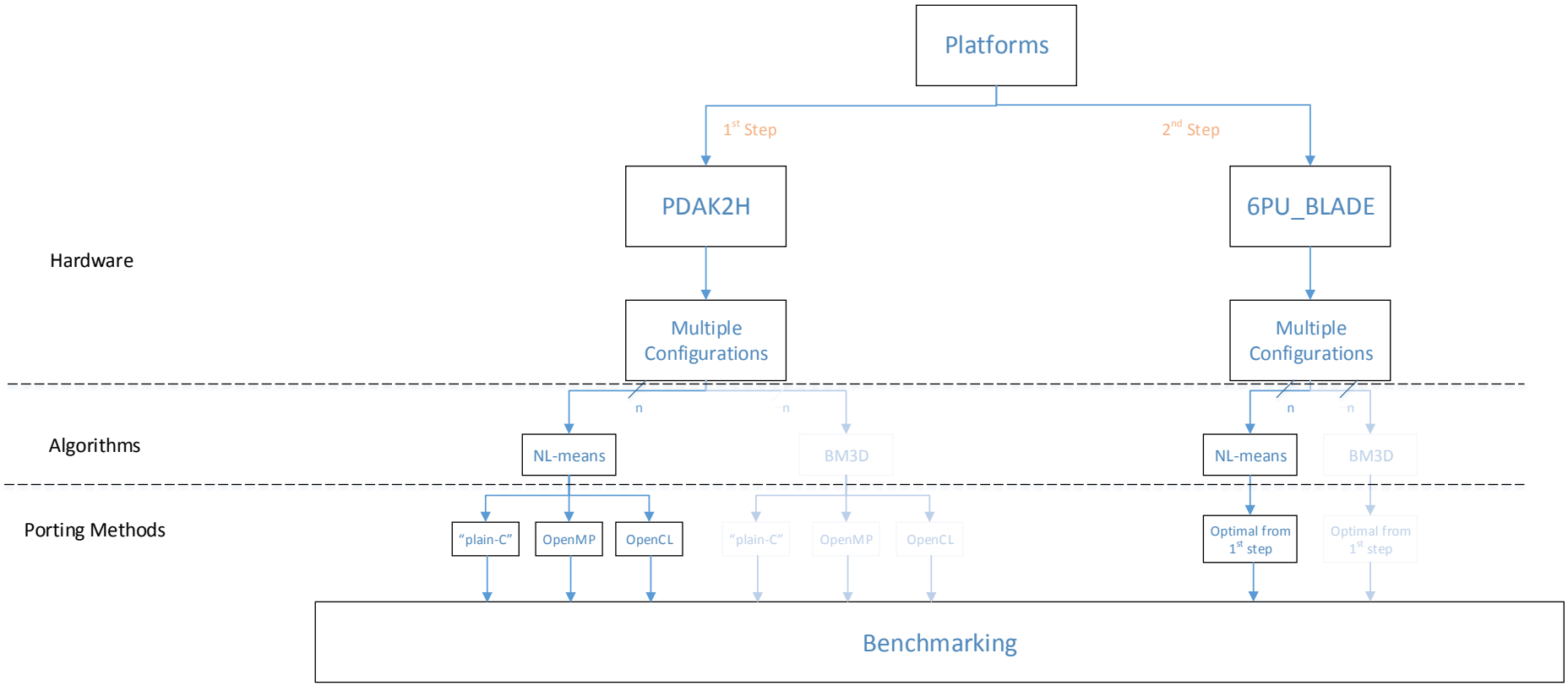

Figure 6.2 Steps and paths of the benchmarking procedure.

6.3. “Plain-C” Experimental Results

In order to investigate the capabilities and the scalability issues of the TI Keystone-II “Hawking” SoC its benchmarking was performed during the preparation phase of the thesis. By varying the number of involved ARMs/DSPs in the target deployment, different configurations of the specific SoC have been created. The following table presents the configurations that have been examined.

Configuration	ARM A15	C66x DSP
1	1	-
2	2	-
3	3	-
4	4	-
5	-	1
6	-	2
7	-	3
8	-	4
9	-	5
10	-	6
11	-	7
12	-	8
13	4	8

Table 6.1 Configurations tested on Keystone-II “Hawking” SoC.

As a first step the manual parallelization of NL-M has been made in “plain-C”. As it is discussed in Chapter 5, NL-M can easily be parallelized by splitting the image into parts. By using the above procedure, the workload can be distributed to all the available processing units since there are no data dependencies.

6.3.1. ARM configurations

Four different configurations (indicated in lines 1-4 in Table 6.1), in which only ARM processors are utilized, have been examined. As it has already been mentioned above, the parallelization has been made manually in order to distribute equally the workload to the available processing units. Moreover, the workload has been offloaded to the available processing units with the help of OpenMP library. A proportional to the number of involved ARM processors decrease in the computation time can be expected. Figure 6.3 illustrates the results that have been obtained for the aforementioned experiments.

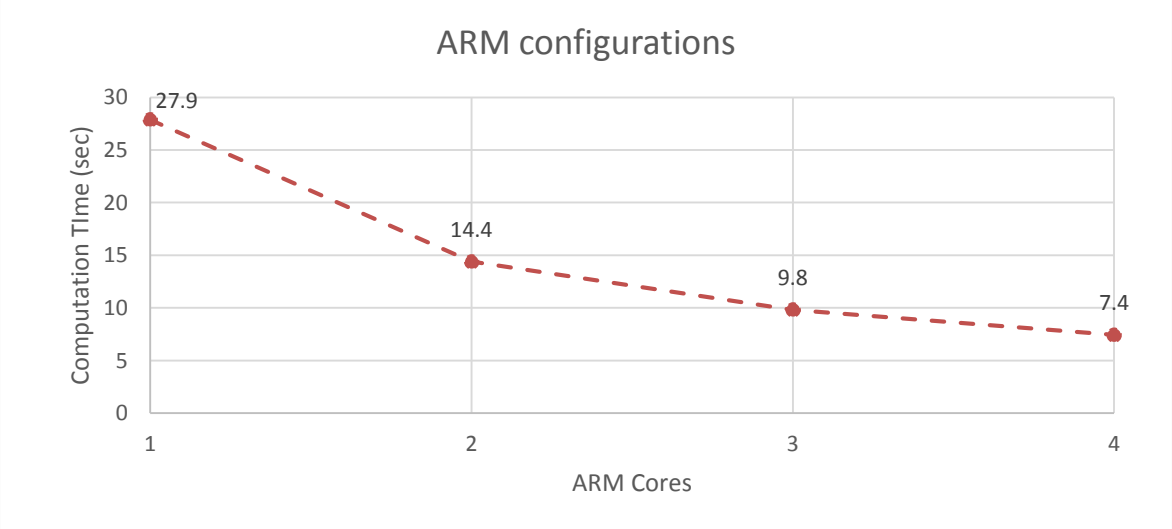

Figure 6.3 NL-M computation time in ARM configurations with 1 – 4 cores

As it can be seen from the above figure, the initial hypothesis was correct since the computation time of the examined application decreases proportionally as the number of available processing units increases. Moreover, Table 6.2 presents the comparison between the measured and theoretical value of the computation time. Figure 6.4 shows that the speedup is close to the theoretical one.

ARM cores	Computation Time (sec)		Difference (sec)	Speedup over single core
	Measured	Theoretical		
1	27.9	-	-	-
2	14.4	14.0	0.4	1.9x
3	9.8	9.3	0.5	2.8x
4	7.4	7.0	0.4	3.8x

Table 6.2 Measured and Theoretical computation times in ARM configurations

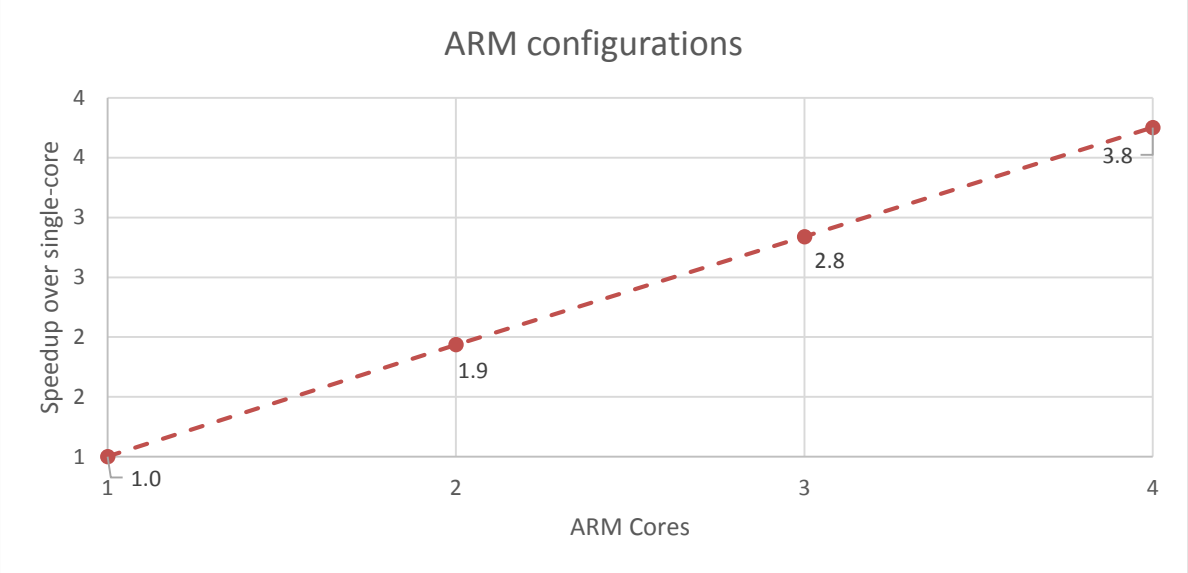

Figure 6.4 Speedup over single-core configuration

6.3.1.1. Further Experimentation

For higher efficiency, pthread library is used for the parallelization of the application. However, the difference between the OpenMP implementation is not significant regarding the measured execution times.

For further experimentation regarding the performance of the ARM Cortex-A15 processor a roofline model is constructed. According to ARM Cortex-A15 specs the processor is able to execute up to 8 SP FLOPs/cycle by executing either two 4-wide NEONv2 FMA or two 4-wide NEON multiply-add (one in each side). Hence, the ARM cortex-A15 subsystem can offer up to 11.2 GFLOPS/Core for Floating Point @ 1.4 GHz.

The maximum theoretical throughput of a memory copy from/to the shared memory to/from a cortex-A15 core, when a 128-bit width bus is used, is:

$$(128 \text{ bits}) / (8\text{bits/byte}) * 1.4 \text{ GHz} = 22.4 \text{ GB/s}$$

From the above, the roofline model for several ARM-configurations of the TI Keystone-II “Hawking” SoC, is constructed. The built-in NEON intrinsics for the ARM Advanced SIMD extension are available when the `-mfpu=neon` compiler option is used.

Figure 6.5 illustrates the roofline model for the single core and 4-core ARM configuration when the NEON instructions are enabled or disabled. Two different values of maximum bandwidth between the cortex-A15 cores and the shared memory are considered.

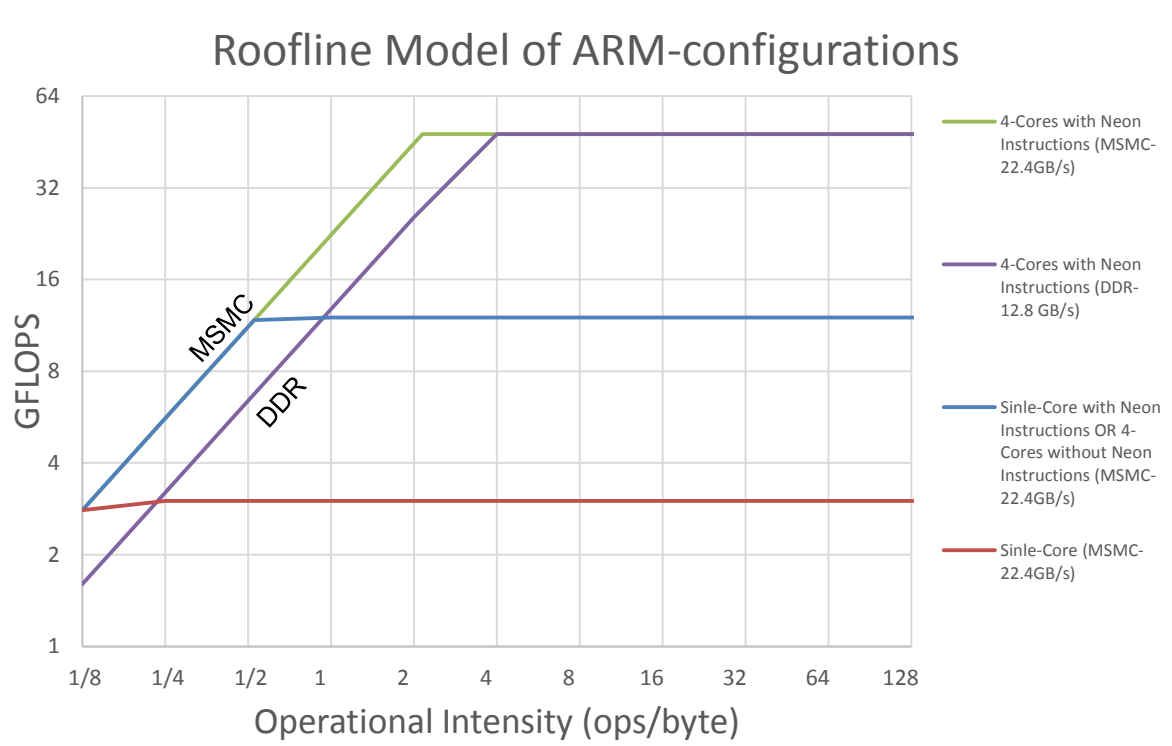

Figure 6.5 TI Keystone-II "Hawking" SoC Roofline Model using different ARM configurations and shared memory

A micro-benchmarking application was developed in order to verify the maximum theoretical performance of the cortex-A15 processor. More specifically the multiply-accumulate `vmlaq_f32` intrinsic is used. The following code snippet presents the kernel of the specific micro-benchmarking. 42 GFLOPs (88% of the theoretical) was the maximum achieved performance.

```
float32x4_t result = vmovq_n_f32 (0);
float32x4_t data;
float32x4_t data2;
for (int i = start; i < end; i += 4) {
    data = vld1q_f32(float32_data + i);
    data2 = vld1q_f32(float32_data2 + i);
    result = vmlaq_f32(result, data, data2);
}
```

6.3.2. DSP configurations

First, the single-core DSP configuration has been tested where several optimizations have been introduced. The multi-core DSP configurations followed. Finally, the roofline model for the 8-core DSP configuration is presented.

6.3.2.1. Single-core optimizations

Several optimizations have been applied in the single core DSP implementation. The C/C++ compiler offers various optimizations via flags. Firstly, `--opt_level3` (or `-O3`) option has been selected. Some of the most important optimizations which are offered by the compiler with this level of optimization are the following:

- Allocates variables to registers
- Expands calls to functions declared inline
- Performs software pipelining
- Performs loop optimizations
- Performs loop unrolling
- Converts array references in loops to incremented pointer form
- Inlines calls to small functions

Moreover, `--opt_for_speed=num` option has been examined. With this option the user specifies the type and the degree of code size of code speed optimization. The `--opt_for_speed=5` option has been selected which enables optimizations geared towards improving the code performance/speed with a high risk of worsening or impacting code size.

Passing information to the compiler via pragmas is also a common optimization technique. Two pragmas have been tested:

- The `MUST_ITERATE(lower_bound, upper_bound, factor)` pragma which specifies to the compiler certain properties of the loop and
- The `UNROLL(factor)` pragma which instructs the compiler to unroll exactly what the programmer wants it to

However, the introduction of both pragmas did not affect the performance of the application, because of the aforementioned options that have been selected.

Finally, the loop tiling optimization technique has been tested. In the following table the performance gain after each examined optimization technique is presented. The total performance gain that has been achieved after the introduction of all the optimization techniques is 72% with respect to the initial computation (82.3 seconds).

Optimization Technique	Computation Time (sec)	Performance gain (%)
None	82.3	-
-O3	29.4	64%
--opt_for_speed = 5	24.6	17%
MUST_ITERATE pragma	24.6	-
UNROLL pragma	24.6	-
Loop Tiling	23.3	5%

Table 6.3 Single-Core DSP Optimizations

6.3.2.2. Multi-core implementation

Similar to the ARM configurations, the algorithm has been parallelized manually in order to distribute the workload equally to a number of DSP units. The workload has been offloaded to the available processing units using the MPM library, which has been discussed in Chapter 2.1. Again, it can be expected that the decrease of the computation time is proportional to the number of involved DSPs. Figure 6.6 illustrates the behaviour of the application in the DSP configurations.

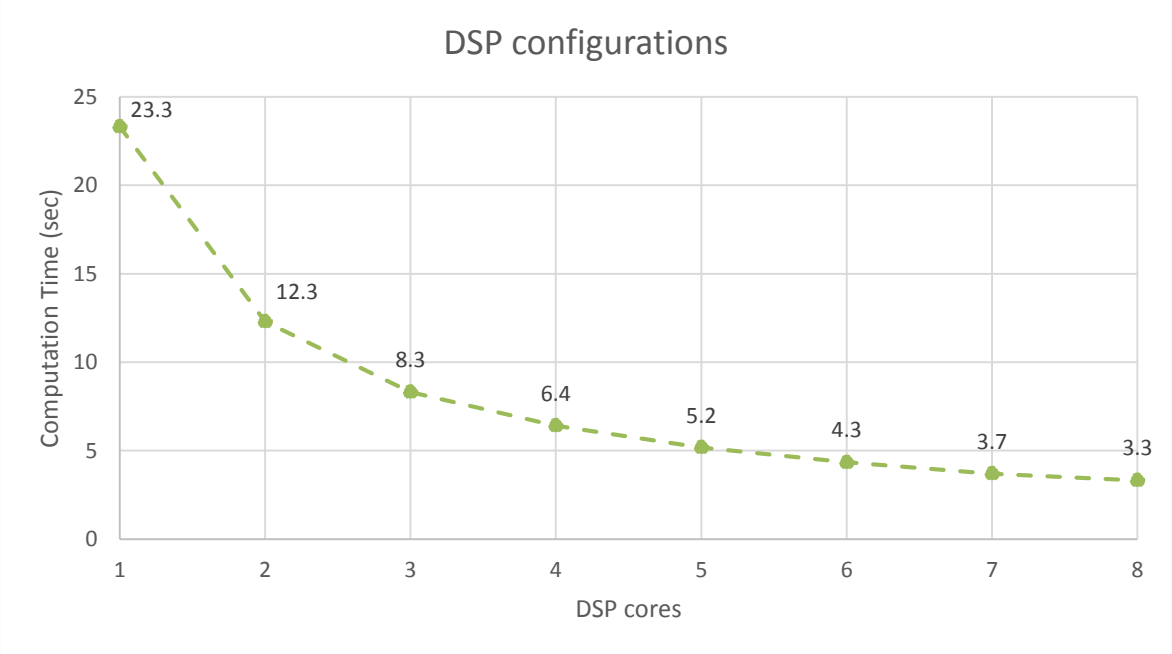

Figure 6.6 NL-M computation time for various DSP configurations

Indeed, the computation time of the examined application decreases proportionally as the number of available processing units increases. Moreover, Table 6.4, presents the difference between the measured and theoretical value of the computation time. It has to be mentioned that the computation times that are presented in Table 6.4 have been measured in the host (ARM) side. In this way, the computation time includes the overhead to start the device (DSP). Moreover, the host has to poll a specific memory address in order to be informed that the computations are completed. An extra overhead is introduced by this procedure. In the next steps of the benchmarking, the computation time is measured by the device and the speedup is closer to the theoretical one (8x).

DSP cores	Measured Computation Time (sec)	Theoretical Computation Time (sec)	Difference (sec)	Speedup over single core
1	23.3	-	-	-
2	12.3	11.7	0.7	1.9x
3	8.3	7.8	0.5	2.8x
4	6.4	5.8	0.6	3.6x
5	5.2	4.7	0.5	4.5x
6	4.3	3.9	0.4	5.4x
7	3.7	3.3	0.4	6.3x
8	3.3	2.9	0.4	7.1x

Table 6.4 Measured and Theoretical computation times in DSP configurations

In Figure 6.7, the comparison between the computation time of the examined application in the DSP and the ARM configurations is given. The difference can be explained by comparing the maximum performance of each processor. The specific DSP cores give up to 19.2 GFLOPS (8 FLOPS/cycle in both sides, 1.2 GHz) while the ARM cores is limited to 11.2 GFLOPS (4 FLOPS/cycle in both sides – only by using the NEONv2 FMA or the NEON multiply-add instructions, 1.4 GHz).

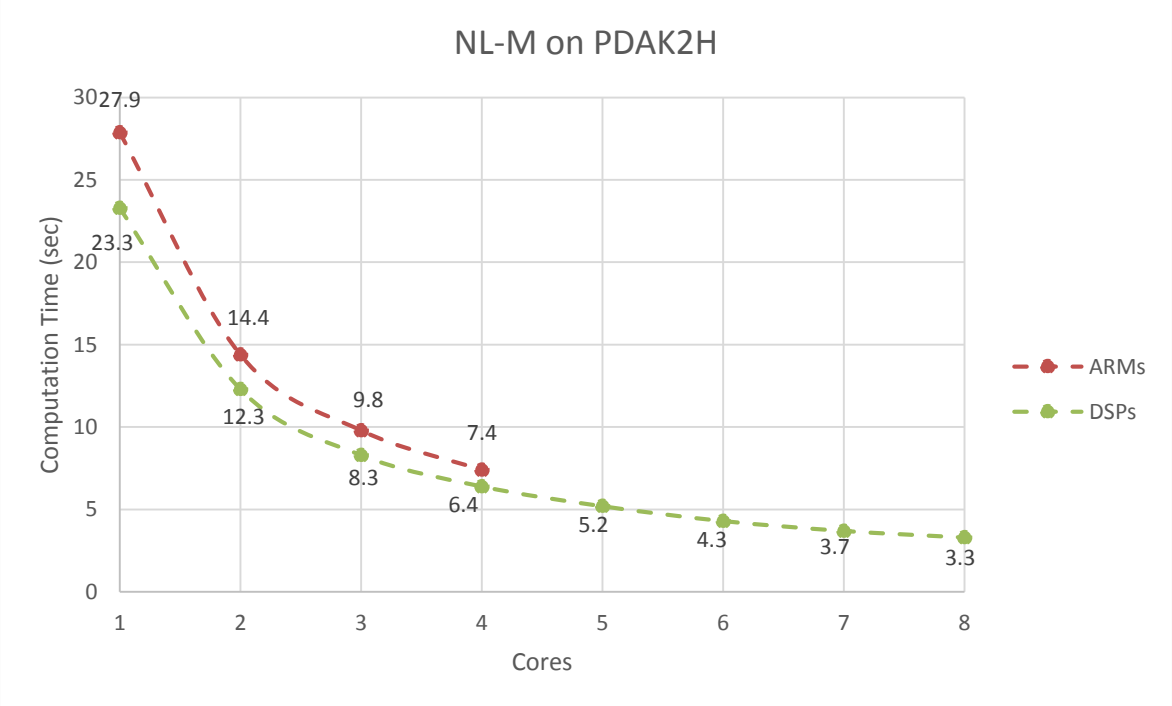

Figure 6.7 ARM - DSP comparison

6.3.2.2.1. Roofline Model

According to TI's datasheet [33] the eight TMS320C66x DSP core subsystems, which are included in the Keystone-II "Hawking" 14 SoC in PDAK2H, can offer up to 19.2 GFLOPS/Core for Floating Point @ 1.2 GHz.

The maximum theoretical throughput of a memory copy from/to the shared memory to/from a C66x device is 16 bytes (128 bits) ∗ 1.2 GHz = 19.2 GB/s.

From the above, the roofline model (Figure 6.8) for the TI Keystone-II “Hawking” SoC, when the 8 DSPs are used, can be constructed.

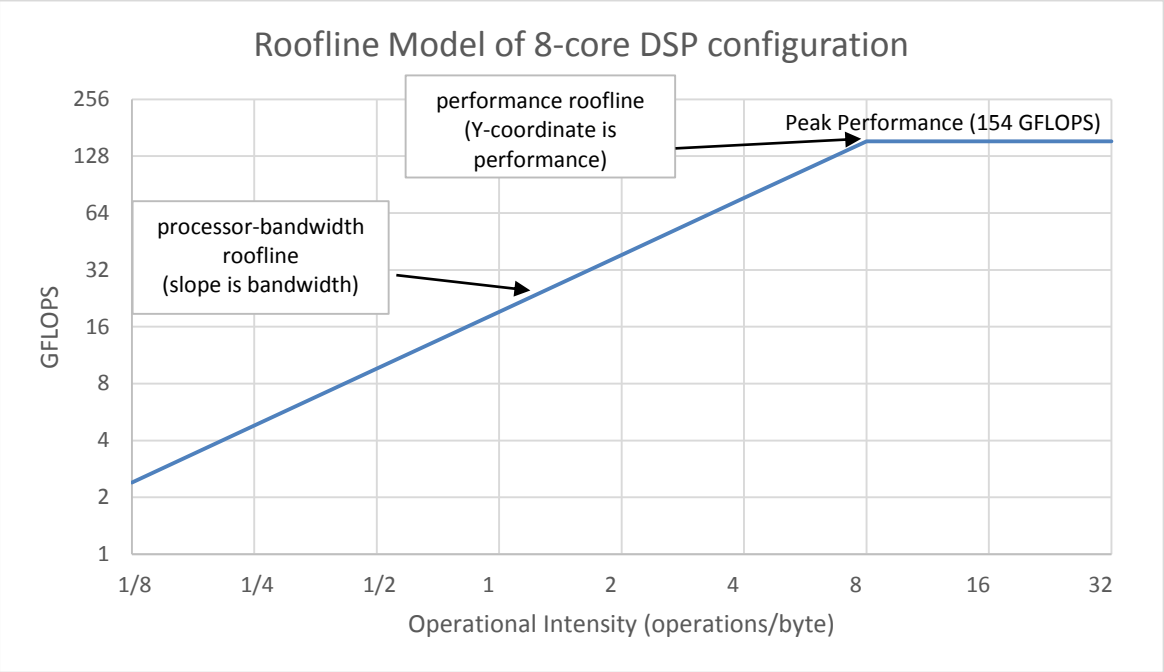

Figure 6.8 TI Keystone-II “Hawking” SoC Roofline Model using 8-core DSP configuration and shared memory

In subsection 5.2 the formula which computes the total number of floating point operations that are needed for denoising a single pixel via NL-M has been given. Hence, for a 704x469 image, using parameters for R and F from subsection 5.2, the total number of floating point operations is:

$$\left(\left(704 - 2 * (R + F)\right) * \left(469 - 2 * (R + F)\right)\right) * \left((2 * R + 1)^2 * (2 * F + 1)^2 * 3 + (2 * R + 1)^2 * 6 \ + 2\right)$$

Regarding the memory accesses if it is assumed that each pixel is read only once due to the available caches, it can be calculated that:

$$N_{READ} = 704 * 469$$

$$N_{WRITE} = \left(704 - 2 * (R + F)\right) * \left(469 - 2 * (R + F)\right)$$

$$N_{TOTAL} = N_{READ} + N_{WRITE}$$

Hence, by changing the size of the window (R) or the patch (F) the operational intensity of the application will change since the total number of memory accesses will change insignificantly with respect to the great change of the floating point operations. Figure 6.9 illustrates the behaviour of the application in the already constructed roofline model.

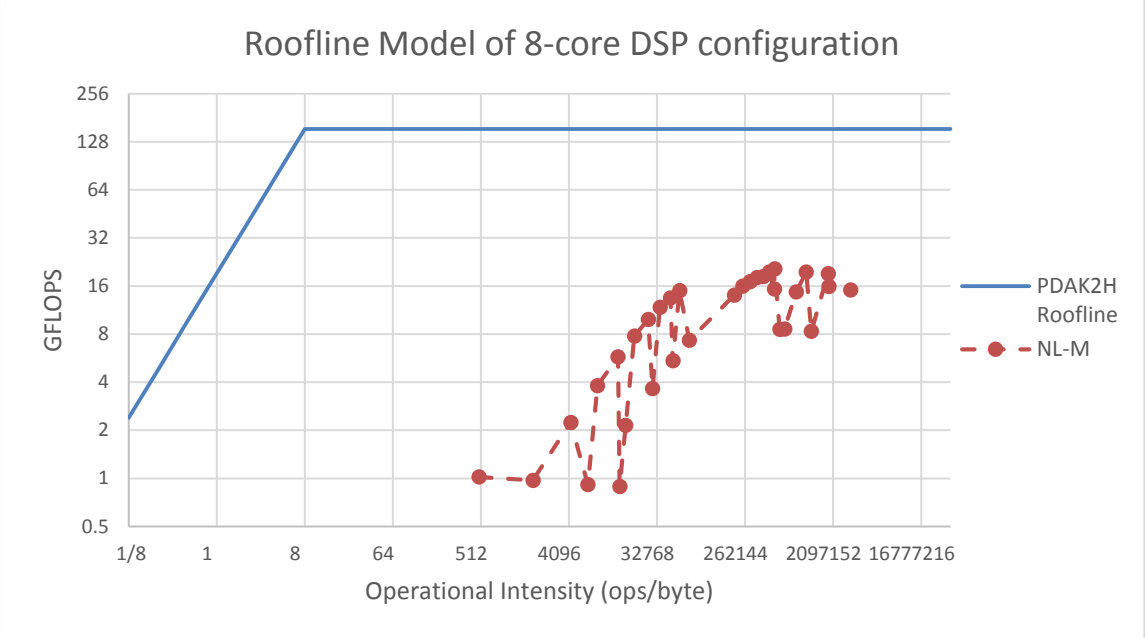

Figure 6.9 NL-M in PDAK2H's roofline model for various sizes of windows and patches

The figure shows that the specific application is bounded by the performance roofline since the examined versions have an operational intensity which is much greater than the limit in which it stops to be bounded by the processor-bandwidth. It can be concluded that with the same operational intensity by maximizing the hardware utilization a greater performance can be achieved. The above can be accomplished by a different implementation of the algorithm which uses more efficiently the functional units. Moreover, Figure 6.9 shows how the size of the window and the patch affects the performance of the application. This happens because with some configurations the cache memory is not used as efficiently as it is used with some others. The following table gives the configurations where the four lowest and the four highest values of GFLOPS have been measured.

Window Size	Patch Size	Operational Intensity	GFLOPS attained
61x61	3x3	50000	0.9
41x41	3x3	25000	0.9
21x21	3x3	7000	1.0
11x11	3x3	2000	1.0
81x81	31x31	7000000	19.1
41x51	31x31	2000000	19.6
61x61	31x31	4000000	19.6
41x41	31x31	2000000	20.5

Table 6.5 Window and Patch size affects application's performance

6.3.2.3. Further Experimentation

Figure 6.10 presents the PDAK2H roofline for the single core and the 8-core configuration. Two different values of maximum bandwidth between the C66x cores and the shared memory are considered. The maximum theoretical bandwidth when a 128-bit width bus is used, is:

$$(128 \text{ bits}) / (8\text{bits/byte}) * 1.2 \text{ GHz} = 19.2 \text{ GB/s}$$

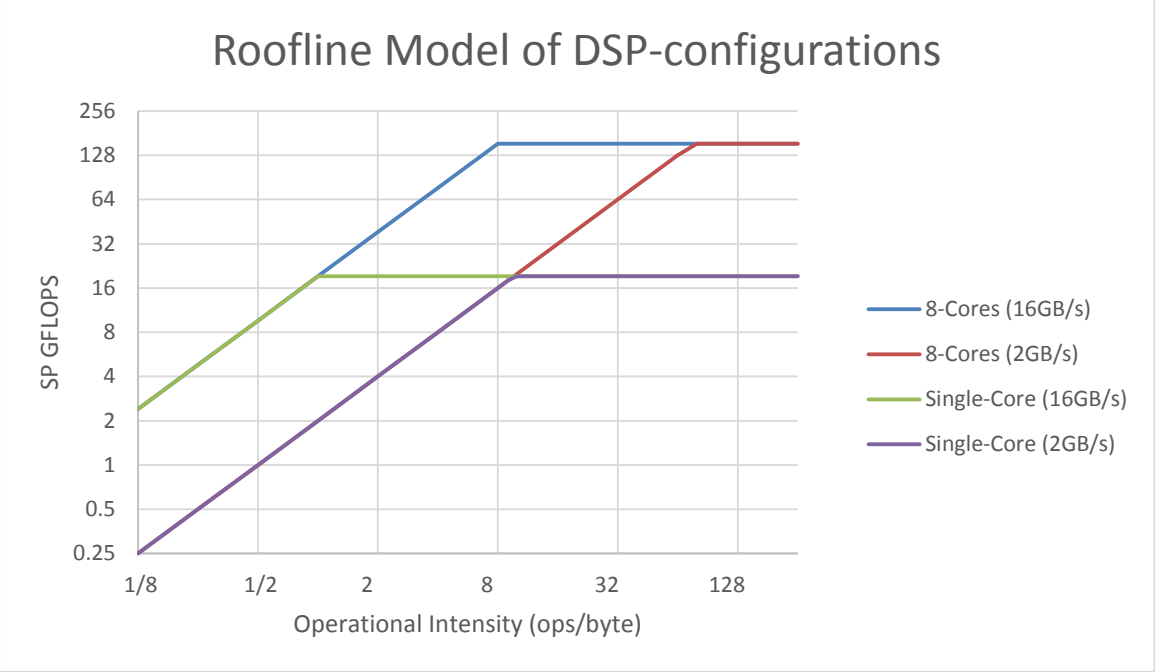

Figure 6.10 TI Keystone-II "Hawking" SoC Roofline Model using different DSP configurations and shared memory

Micro-benchmarking was performed in order to verify the maximum theoretical performance of the C66x DSP core. The C66x DSP subsystem of both available SoCs (Keystone-I and Keystone-II) was benchmarked with the LINPACK micro-benchmark for DP FLOPs. 32 GFLOPS (DP) are achieved for both SoCs.

Moreover, a micro-benchmarking application was developed in order to verify the maximum theoretical bandwidth between the C66x DSP core and the shared memory. It has to be mentioned that a DMA needed to be used in order to achieve the maximum bandwidth.

As a next step, the cache configuration has been taken into account. By default the MSMC memory is cacheable and prefetchable. By configuring MSMC as non-cacheable and non-prefetchable the performance of the application was decreased more than 10x.

As it is mentioned above, the initial assumption was that the each pixel is read only once. In the following figure the behaviour of NL-M it is presented by assuming:

1) The caches are not used
2) Only the most inner loop (F-loop) is cached
3) Only the two most inner loops are cached (FF-loop, patch window)
4) Only the three most inner loops (RFF-loop) are cached
5) Each pixel is read only once (perfect caching)

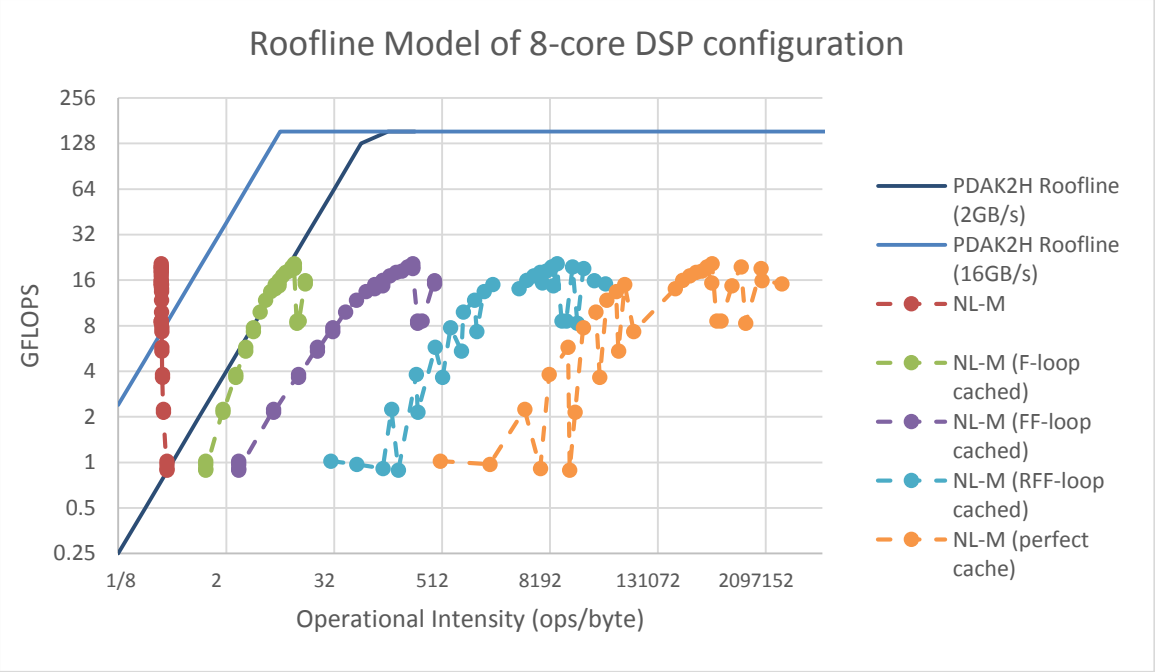

Figure 6.11 NL-M in PDAK2H's roofline model for various sizes of windows and patches and different configurations

It is clear that the behaviour of the application and how close the application is to the roofline model is highly depended on the amount of data that are stored in the cache. Another parameter that is taken into account is the maximum achievable bandwidth between the C66x core and the MSMC memory. As it is mentioned above the maximum theoretical bandwidth can be achieved only if the DMA is used. It seems that for a configuration where the maximum achievable memory bandwidth is limited to 2GB/s and only the most inner loop (F-loop) is able to be cached, the application is really close to the expected roofline. However, there are still questions regarding the specific behaviour of NL-M.

During the investigation regarding the C66x DSP performance several bottlenecks have been identified.

All the divisions are replaced from the kernel by the multiplication of the inverse number. Moreover, it was noticed that the `expf(-1.0f*base)` function introduces a significant overhead. Hence, it was replaced by its approximation using the Taylor series of order 3:

$$e^{-x} \approx 1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!}$$

In this way the number of floating point operations can be determined more accurate. The final implementation is presented in the following code snippet.

```
float *in = (float *)(SHM_BASE_ADDRESS);
float *out = in + SIZEX*SIZEY;
const float S = 15/255.0f;
const float H = 0.40f*S;
const float HH = H*H;
const float INV_HH = 1.0f/HH;
float dd, w, base;
int px, py, qx, qy, fx, fy;
int rows_per_dsp = (SIZEY-(R+F)*2) / NR_DSPS;
int start_row = (DNUM)*rows_per_dsp+R+F;
int end_row = start_row+rows_per_dsp;
if(end_row > SIZEY-R-F)
    end_row = SIZEY-R-F;
for(py=start_row; py<end_row; py++) {
    for(px=R+F; px<SIZEX-R-F; px++) {
        float Cp = 0;
        float sum = 0;
        for(qy=-R; qy<=+R; qy++) {
            for(qx=-R; qx<=+R; qx++) {
                float d = 0;
                for(fy = -F; fy <= +F; fy++) {
                    for(fx = -F; fx <= +F; fx++) {
                        float pix_p = in[(py +      fy)*SIZEX + px +      fx];
                        float pix_q = in[(py + qy + fy)*SIZEX + px + qx + fx];
                        float delta = pix_p - pix_q;
                        d += delta * delta;
                    }
                }
                dd = (1.0f / ((2*F+1) * (2*F+1))) * d - 2.0f*S*S;
                base = dd > 0.0f ? (dd*INV_HH) : (0.0f);
                w = 1.0f - base + (base*base)*0.5f - (base*base*base)*0.17f;
                w = w < 0.0f ? (0.0f) : (w);
                Cp += w;
                sum += in[(py + qy)*SIZEX + px + qx] * w;
            }
        }
        out[py*SIZEX+px] = (1.0f/Cp) * sum;
    }
}
```

Hence, for a 704x469 image, using parameters for R and F from subsection 5.2, the total number of floating point operations is:

$$((704 - 2 * (R + F)) * (469 - 2 * (R + F))) * ((2 * R + 1)^2 * (2 * F + 1)^2 * 3 + (2 * R + 1)^2 * 14 + 2)$$

It has to be mentioned that the after the aforementioned modifications on the NL-M code a significant speedup was achieved. Especially in cases where the patch size is small and the window size big enough the speedup is more than 40x. Table 6.6 shows the achievable speedup for different configurations of patch and window sizes.

Window Size	Patch Size	GFLOPS attained		Computation Time		Speedup
		Original	Modified	Original	Modified	
11x11	3x3	1.02	26.16	1.24	0.06	20.7x
21x21	3x3	0.97	34.45	4.57	0.16	28.5x
41x41	3x3	0.91	43.29	17.19	0.45	38.2x
61x61	3x3	0.89	47.46	36.08	0.84	42.9x
21x21	11x11	7.78	18.09	6.19	2.72	2.3x
41x41	31x31	20.53	26.14	59.85	47.13	1.3
81x81	41x41	15.14	19.27	445.97	350.91	1.3x
61x61	41x41	15.88	19.15	263.63	219.04	1.2x

Table 6.6 Comparison between original and modified NL-M.

It is noticed that as the size of the patch increases the speedup decreases. This happens because the applied optimizations do not affect the most inner loop over the patch size. Moreover Table 6.6 indicates that the `expf()` function should not be counted as a single floating point operation since its replacement by a number of other floating point operations gives not only a great speedup but

also an increase to the attained GFLOPS. In Figure 6.12 the modified NL-M is plotted in the 8-DSP configuration roofline.

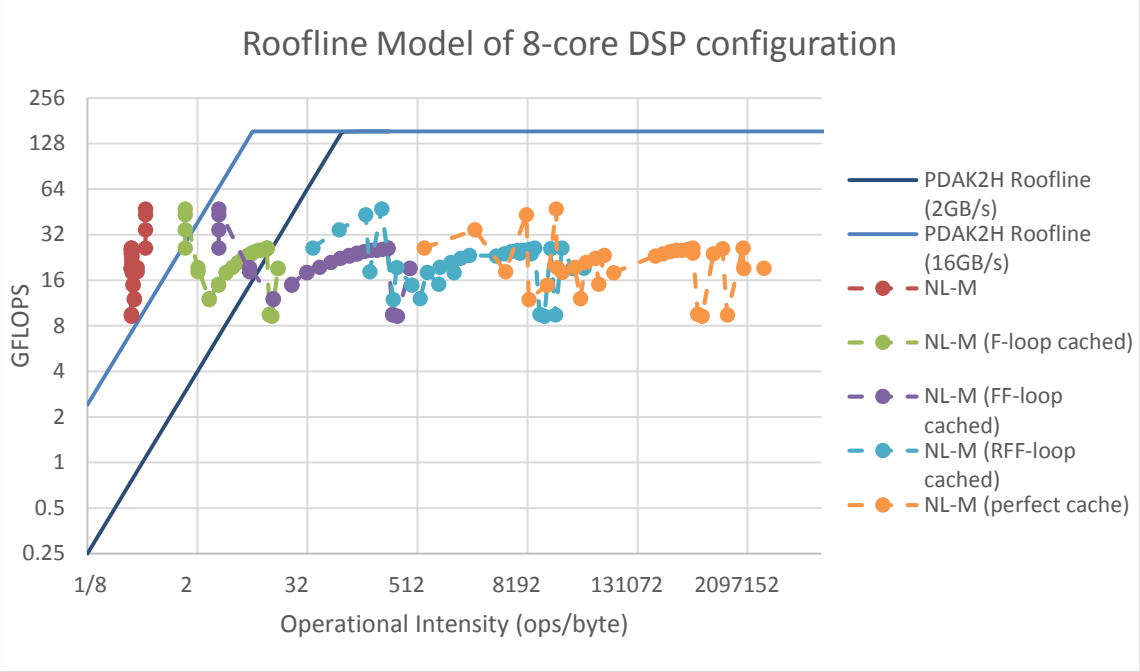

Figure 6.12 Modified NL-M in PDAK2H's roofline model for various sizes of windows and patches and different configurations.

As it is discussed above, after the adjustments in the code the achievable performance is much higher than the previous. However, again it is not easy to justify the difference of the expected roofline.

Similar experiments have been performed for the ARM A-15 cores. The results are not presented because of their similarity with those for the DSP configurations.

6.3.3. Heterogeneous configurations

As a next step the combination of both types of processors (ARMs/DSPs) has been examined. All the available processing units (four ARMs and eight DSPs) have been involved.

In the first experiment, which has been performed in the heterogeneous configurations, the image has been divided equally to all (12 in number) available processing units. Hence, every processing unit is responsible for the process of the 1/12 of the input data. Table 6.7 presents the computation time of the complete application and the computation time of the separate parts which were mapped in the ARM and in the DSP side.

Application	Computation time (sec)
Complete (max(ARM-part, DSP-part))	2.4
ARM-part	2.4
DSP-part	2.3

Table 6.7 Heterogeneous Configuration – Equal workloads

From the above table it can be concluded that splitting the image in equal parts and distribute equal workloads to ARMs and DSPs a lower execution time is achieved with respect to the 8-DSP configuration. However, a different workload allocation can be calculated by taking into consideration the performance of each processor. Figure 6.7 indicates that the DSP single core implementation achieves a ~1.2 times lower computation time than the one with the ARM processor. The following formulas indicate that the theoretical optimal workload allocation is 29% to the ARM-side and 71% to the DSP side.

$$W_{DSP} = 8 * w_{DSP} = 8 * (1.2 * w_{ARM}) = 8 * \left(1.2 * \frac{W_{ARM}}{4}\right) = 2.4 * W_{ARM}$$
$$W_{ARM} + W_{DSP} = 100\%$$

Where: $w_{DSP}$and $W_{DSP}$are the workloads for the one DSP core and for the complete DSP side, respectively and $w_{ARM}$and $W_{ARM}$are the workloads for the one ARM core and for the complete ARM side, respectively.

By performing a series of experiments, it was shown that the optimal, with respect to computation time, division of the workload is very close to the theoretical. Hence, it can be concluded that the performance of the available processing units affects the optimal workload allocation. The above is presented in Table 6.8. Figure 6.13 demonstrates how the workload allocation assigned to the DSP-side, affects the execution time of the application.

Processor Type	Workload percentage	Computation time (sec)
ARM-side	32%	2.3
DSP-side	68%	2.3

Table 6.8 Heterogeneous Configuration - Different workloads

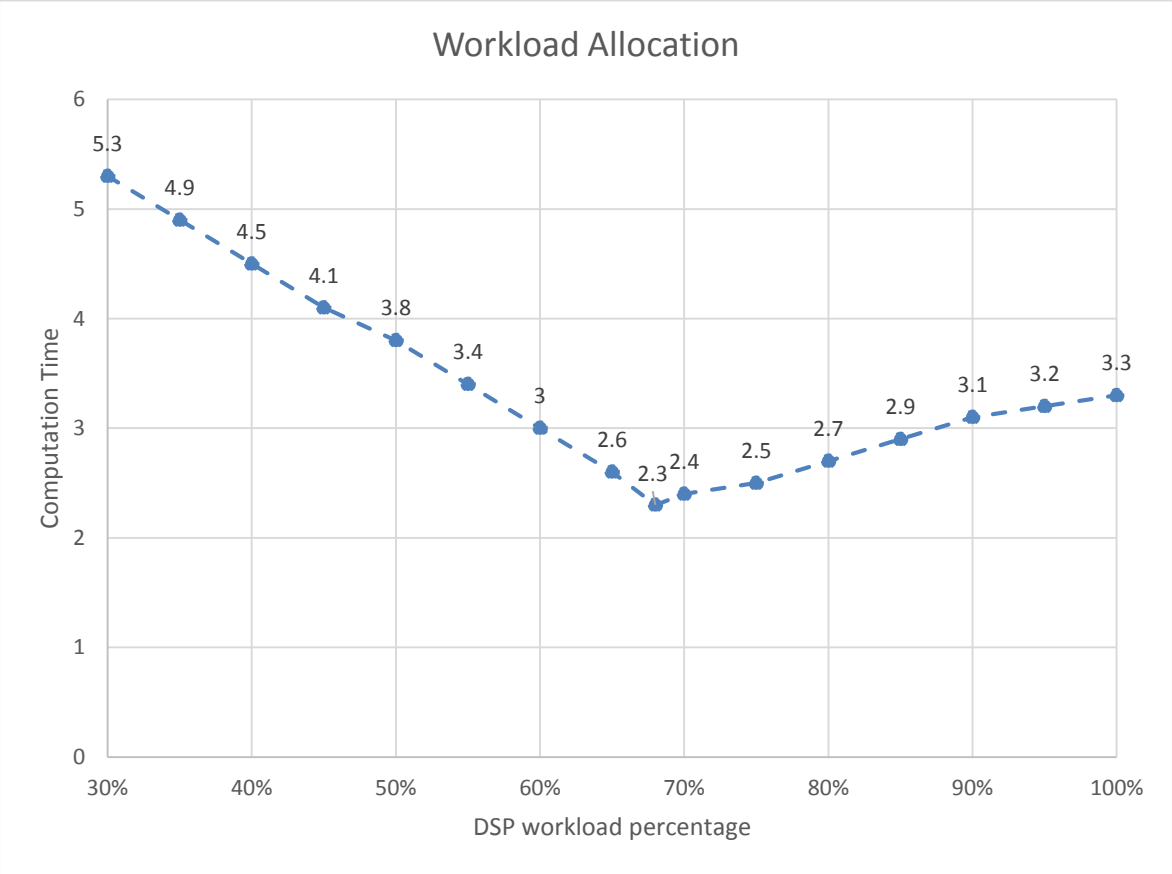

Figure 6.13 Execution time affected by workload allocation

The computation time has been measured for a variety of images in order to investigate how the size of the input data affects the optimal division of the workload among the processors. The following table presents the results for different sizes of images.

Image Size	Processor Type	Workload allocation	Computation time (sec)
512x512	ARM-side	32%	2.3
	DSP-side	68%	2.3
704x469	ARM-side	32%	3.1
	DSP-side	68%	3.1
1024x768	ARM-side	33%	7.6
	DSP-side	67%	7.6

Table 6.9 Optimal workload distribution for different sizes of input image

Table 6.10 presents the relative decrease of the computation time after the optimal division of the workload in comparison with the DSP configuration which uses all the available DSPs.

	Computation Time		
Image Size	DSP only configuration	Heterogeneous configuration	Performance gain (%)
512x512	3.3	2.3	31%
704x469	4.6	3.1	32%
1024x768	11.4	7.6	33%

Table 6.10 Comparison between DSP and Heterogeneous configurations

It can be concluded, that the optimal division of the workload between the available processing units for different sizes of images is also very close to the theoretical one.

6.4. OpenMP and OpenCL Experimental Results

During the thesis preparation phase, an initial investigation has been made regarding the porting of the NL-M algorithm with OpenMP and OpenCL programming methods. The following table presents the results that have been obtained. It is clear that OpenCL employment performs much better than OpenMP.

Image Size	Programming Method	Execution time (sec)
512x512	OpenMP	14.2
	OpenCL	2.7
	“plain-C”	3.3
704x469	OpenMP	18.2
	OpenCL	4.8
	“plain-C”	4.6
1024x768	OpenMP	44.3
	OpenCL	9.2
	“plain-C”	11.4

Table 6.11 OpenMP and OpenCL comparison

It has to be mentioned that the execution time of OpenCL excludes the time that is needed for the host (ARM cores) to initialize the device (almost 5 seconds) and retrieve the data (less than 1 msec). However, even if we include the above times to the total execution time, OpenCL implementation behaves much better than the corresponding implementation in OpenMP. The above can be explained by the significant overhead which is caused by the data synchronization between the host (ARM) and target (DSP) device. TI has implemented an API to reduce the overhead of offloading target regions. The TI-provided functions can be used to allocate and free contiguous segments of memory that may be accessed by all ARM and DSP cores. The aforementioned functions have been used for the image sizes that are presented in Table 6.11. However, the results were not the expected ones since the execution time decreased for less than a second (the specific function could be useful for bigger images).

Moreover, OpenCL employment seems to performs better (or almost equal) than the manual parallelization using “plain-C”. However, as mentioned above, 5 seconds are needed to initialize the device.

6.4.1. Vector Addition using “plain-C”, OpenMP and OpenCL

In order to investigate further the differences between the three programming methods a simpler application as a vector addition was used. Figure 6.14 presents how the size of the vectors affects the execution time of the application in each implementation.

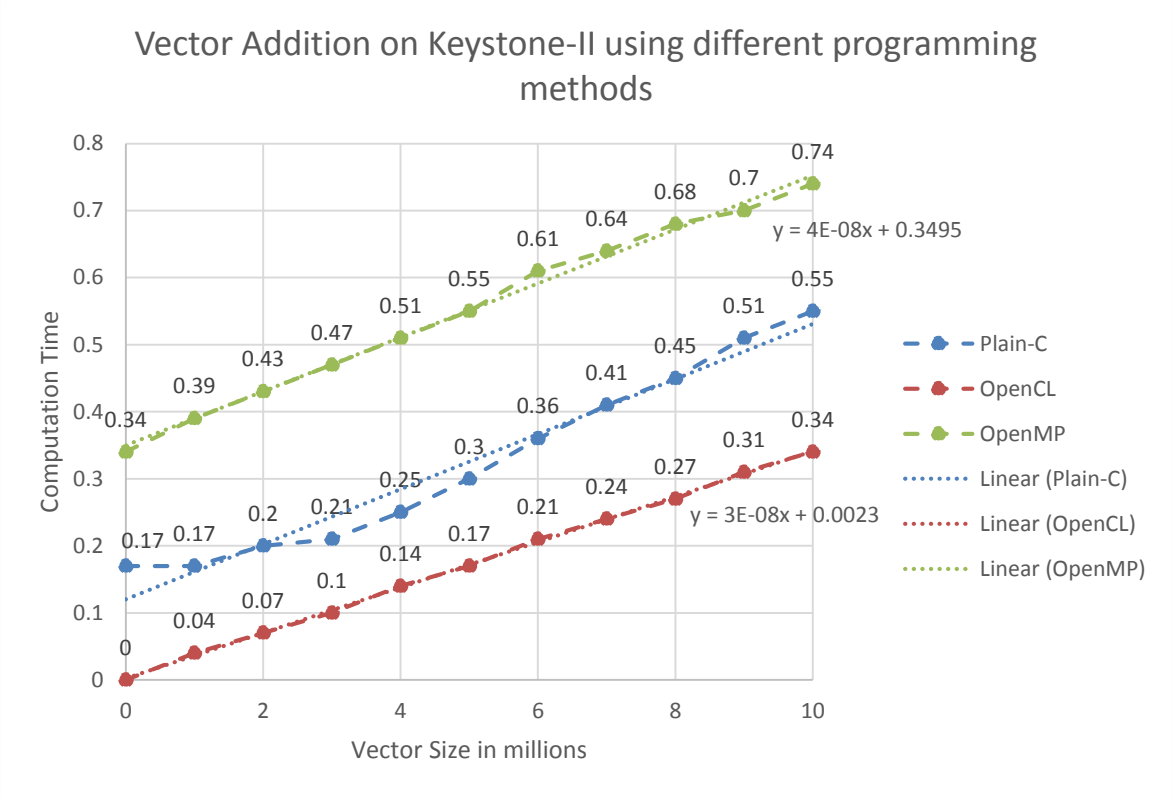

Figure 6.14 Vector addition using "plain-C", OpenMP and OpenCL

The slope of all three line is almost equal and only the offset is different. The OpenCL deployment is the only one without a significant offset at zero vector-size. By measuring the time on the DSP side, it was shown that the offset in the “plain-C” implementation is created due to the overhead that has been discussed in 6.3.2.2. Finally, in OpenMP it has been tried to decrease the computation time of the vector addition by removing the for-loop which is responsible for the addition. The test indicated that the most of the execution time is not spent in the actual execution of the addition but in other tasks. A more detailed investigation was made regarding the behaviour of OpenMP implementation during the next steps of the assignment.

The main remark from the preparation phase, regarding the programming method comparison, is the behaviour of the OpenMP implementation. It was discovered that the observed overhead is introduced because of the inclusion of several header files and functions in the target application. In order to include a header file in this version of OpenMP the specific header file or function needs to be mapped in the available device. Based on [34] via the ‘declare target’ construct the user is able to specify variables/functions/header files that need to be mapped to the device. The following code snippet [34] shows an example of the above.

```
#pragma omp declare target
/* There must be a host and accelerator target definition for this function */
void DSPF_sp_fftSPxSP(int N,
                      float *x, float *w, float *y,
                      unsigned char *brev,
                      int n_min, int offset, int n_max);
#pragma omp declare target end

void dsplib_fft(int N, int bufsize,
                float* x, float* w, float *y,
                int n_min, int offset, int n_max)
{
    #pragma omp target map(to: N, x[0:bufsize], w[0:bufsize], \
                               n_min, offset, n_max) \
                        map(from: y[0:bufsize])
    {
        DSPF_sp_fftSPxSP (N, x, w, y, 0, n_min, offset, n_max);
    }
}
```

By removing the specific constructs both tested applications (NL-M and vector addition) in OpenMP showed the same behaviour with the other two implementations (OpenCL and “plain-C”).

6.4.2. Overall Comparison

OpenMP, OpenCL and “plain-C” were compared by porting two different applications in the available DSP cores. The following remarks have been conducted:

Regarding the computation time, none of the examined programming methods shows a great advantage over the others. Only the OpenCL implementation shows in some cases a slightly better performance than the other two programming methods. A different configuration of the DSPs from the compiler is most probably the reason of this difference. However, a more detailed investigation needs to be done. It has to be mentioned that OpenCL includes a great overhead at the start time due to the initialization that needs to be done. Some of the processes that are involved in this step are the necessary memory allocation, kernel configuration and loading, getting device information etc.

“plain-C” offers much higher flexibility but most of the times manually developing parallel code is a time consuming, complex and error-prone procedure. On the other hand parallelizing an application by using OpenCL or OpenMP is a much easier procedure. The data are distributed among the available cores and they are transferred back to the host automatically. Also, there is no need for polling and waiting for a notification from the target. Among the two examined automated programming methods OpenMP is easier to be used since the user should identify only the opportunities for parallelism and add the necessary directives. Moreover, for the specific SoC and the supported libraries, the user is able to select the number of the available DSPs only via OpenMP (and of course “plain-C”) and not via OpenCL.

6.5. Analytical Performance Models

The main measurement unit which is used in the following analysis is the computation time of a single DSP, t_{comp}^{DSP}.

The analytical performance models are developed for three different configurations:

1. Single Keystone-II Hawking SoC (4xARMs + 8xDSPs)
2. PDAK2H (Keystone-II Hawking (4xARMs+8xDSPs) + 2xShannon (2x8DSPs))
3. ATCA-TK2-6PU Blade (6xPUs)

Assuming an ideal parallelization where the application is distributed equally among the available DSPs, the computation time of the aforementioned configurations is computed as follows:

Computation time of Single Keystone-II Hawking SoC: $t_{comp}^{HWK} = \frac{t_{comp}^{DSP}}{8}$

Computation time of Single Keystone Shannon SoC: $t_{comp}^{SHN} = \frac{t_{comp}^{DSP}}{8}$

Computation time of PDAK2H (or PU): $t_{comp}^{PU} = \frac{t_{comp}^{HWK}(=t_{comp}^{SHN})}{3} = \frac{\frac{t_{comp}^{DSP}}{8}}{3} = \frac{t_{comp}^{DSP}}{24}$

Computation time of ATCA-TK2-6PU Blade: $t_{comp}^{6PU\_BLADE} = \frac{t_{comp}^{PU}}{6} = \frac{\frac{t_{comp}^{DSP}}{24}}{6} = \frac{t_{comp}^{DSP}}{144}$

It has to be mentioned that in all the following models the time needed to divide the data according to the number of available processing units was not taken into account.

6.5.1. Single Keystone-II Hawking SoC

In the following experiments, in addition to the computational cost, the necessary data transfer time between the host (ARM side) and the device (DSP side) needs to be taken into account. Hence, the execution time then can be written as:

$$t_{exec}^{HWK} = t_{comp}^{HWK} + t_{trans}^{HWK}$$

The shared memory (MSMC) between ARMs and DSPs can been used for data storage. Hence, t_{trans}^{HWK} can be obtained by considering the required time to transfer data to/from the shared memory from/to the host (ARM side). Hence,

$$t_{trans}^{HWK} = t_{\mathbf{MSMC \rightarrow ARM}} + t_{\mathbf{ARM \rightarrow MSMC}}$$

Moreover, $t_{\mathbf{MSMC \rightarrow ARM}}$ and $t_{\mathbf{ARM \rightarrow MSMC}}$ is the time that is needed to transfer data from (to) the shared memory to (from) the host and it can be calculated as follows:

$$t_{\mathbf{MSMC \rightarrow ARM}} = \frac{N}{B_{MSMC \rightarrow ARM}}$$
$$t_{\mathbf{ARM \rightarrow MSMC}} = \frac{N}{B_{ARM \rightarrow MSMC}}$$

N: Size of transferred data
$B_{MSMC \rightarrow ARM}$: Bandwidth of a memory copy from MSMC to ARM
$B_{ARM \rightarrow MSMC}$: Bandwidth of a memory copy from ARM to MSMC

Similar performance models can be constructed for the case where the DDR memory is used for data storage and the data transfer time will be larger, due to smaller bandwidth between ARM and DDR. The above happens when the data do not fit in the MSMC. However, an alternative could be to divide the image to smaller chunks in order to use again the shared (MSMC) memory for data storage. A more detailed investigation needs to be done for the specific alternative.

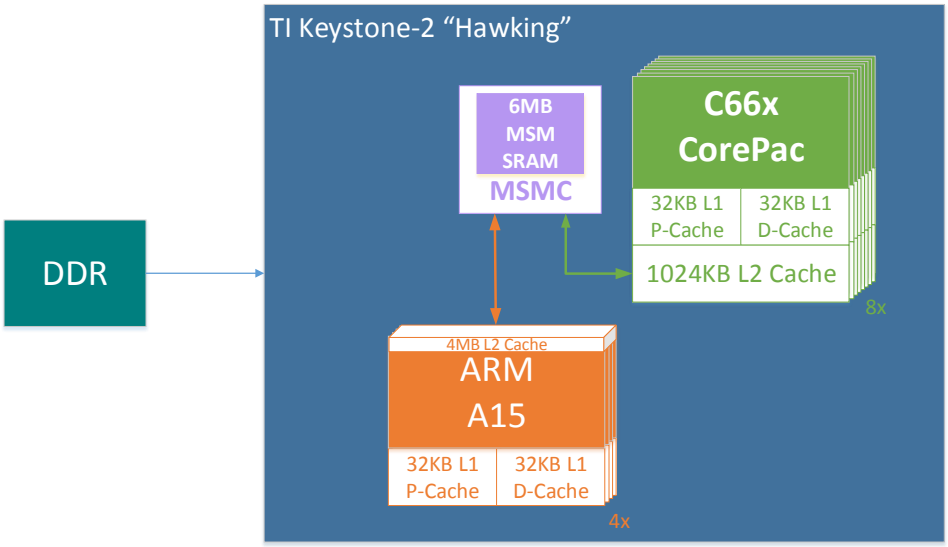

Figure 6.15 TI Keystone-II Hawking SoC Model

According to Keystone-II Hawking SoC's specifications, the theoretical values of the bandwidths that are investigated in the specific analytical performance model are:
$B_{ARM \rightarrow MSMC} = B_{MSMC \rightarrow ARM} = 22.4GB/s$ and $B_{ARM \rightarrow DDR} = B_{DDR \rightarrow ARM} = 12.8GB/s$

However, Table 6.12 shows the difference between the theoretical and measured bandwidths between the host and the MSMC/DDR. Since no DMA or intrinsics have been used for the memory copy between the ARM and the specified memory the maximum theoretical bandwidth cannot be achieved. This happens because the copy does not use the complete width of the bus. In our case, where the `memcpy(3)` function is used for the data transfer, only the 32-bit of the bus width are

used. Also, an extra overhead is introduced because the input data are not stored in the memory of the A15 core but in DDR. Also, it can be noticed that reading back the data from the memory to the ARM is slower than writing them. According to TI's Datasheet this is explained by the fact that a *load* operation needs more cycles than a *store*.

	ARM->MSMC		MSMC->ARM		ARM->DDR		DDR->ARM	
	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured
Transfer Time (msec)	0.04	2.5	0.04	6.3	0.08	2.6	0.08	10
Bandwidth (GB/s)	22.4	0.4	22.4	0.2	12.8	0.4	12.8	0.1

Table 6.12 Theoretical and measured times of a 1MB memory copy between host (ARM of "Hawking") and "Hawking" MSMC/DDR memory.

The same experiments have been performed for a memory copy of 4MB of data and the results were similar. Hence, it was decided to use the above measured bandwidths for the calculation of the theoretical transfer times.

6.5.1.1. Verifications

Table 6.13 and Table 6.14 present the comparison between the theoretical and the measured execution times of NL-M in the single and in the 8-DSP configuration respectively, for four different sizes of input image. It needs to be noticed that the transfer times in both configurations are equal since in both cases the complete amount of input data has to be transferred from/to the ARM to/from the MSMC/DDR memory.

SIZE (memory)	512x512 (MSMC)		704x469 (DDR)		1024x768 (DDR)		4736x3491 (DDR)	
Time (msec)	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured
t_{exec}^{HWK}	1208.4	1198.7	1293.2	1281.0	6759.4	6719.4	71546.4	70686.5
t_{comp}^{DSP}	1190.0		1270.0		6680.0		69880.0	
t_{trans}^{DSP}	9.2	8.7	11.6	11.0	39.7	39.4	833.2	806.5
$t_{\text{ARM}\rightarrow\text{MEM}}$	2.6	2.4	3.3	3.1	8.2	7.1	171.9	139.7
$t_{\text{MEM}\rightarrow\text{ARM}}$	6.6	6.3	8.3	7.9	31.5	32.3	661.3	666.8

Table 6.13 Theoretical and measured values of TI Keystone-II "Hawking" SoC analytical performance model for a single-DSP configuration.

Figure 6.16 presents the execution time in the 8-DSP configuration which includes the two memory transactions from and to the host (ARM) to and from the MSMC. Figure 6.17 and Figure 6.18 presents the relevant results when the DDR memory is used. If the input and the output data fit in the MSMC memory then the 6MB MSMC is the memory which is used for the data storage. The following formula shows the relationship between the selection of the memory and the input data.

$$\text{MEM} = \begin{cases} \text{MSMC}, & \text{SIZE} * \text{sizeof(float)} * 2 \leq 6\text{MB} \\ \text{DDR}, & \text{else} \end{cases}$$

SIZE (memory)	512x512 (MSMC)		704x469 (DDR)		1024x768 (DDR)		4736x3491 (DDR)	
Time (msec)	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured
t_{comp}^{DSP} *	1190		1270		6680		69880	
error (theoretical-measured)	-0.4%		-0.4%		-0.4%		+0.2%	
t_{exec}^{HWK}	158.0	158.7	170.4	171.0	874.7	999.4	9568.2	9546.5
t_{comp}^{HWK}	148.8	150.0	158.8	160.0	835.0	839.0	8735.0	8740.0
t_{trans}^{HWK}	9.2	8.7	11.6	11.0	39.7	39.4	833.2	806.5
$t_{\text{ARM}\rightarrow\text{MEM}}$	2.6	2.4	3.3	3.1	8.2	7.1	171.9	139.7
$t_{\text{MEM}\rightarrow\text{ARM}}$	6.6	6.3	8.3	7.9	31.5	32.3	661.3	666.8

*t_{comp}^{DSP} is the measured computation time in the single DSP configuration

Table 6.14 Theoretical and measured values of TI Keystone-II "Hawking" SoC analytical performance model for an 8-DSP configuration.

It can be seen that the theoretical execution time is almost equal to the theoretical one in all the cases. However, it needs to be noticed that the theoretical transfer times were calculated by using the measured bandwidth of the 1MB memory copy.

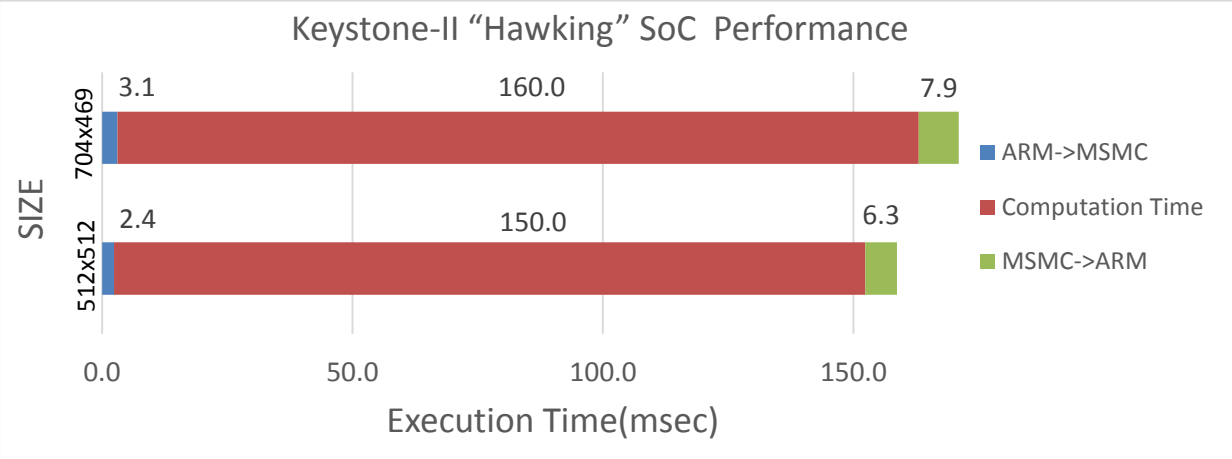

Figure 6.16 TI Keystone-II “Hawking” SoC Performance using MSMC (512x512 and 704x469 images).

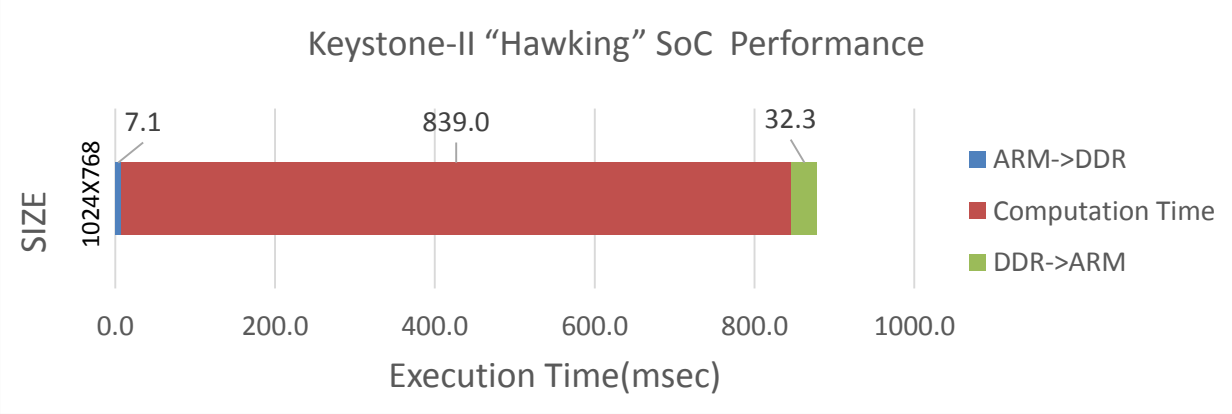

Figure 6.17 TI Keystone-II “Hawking” SoC Performance using DDR (1024x768 image).

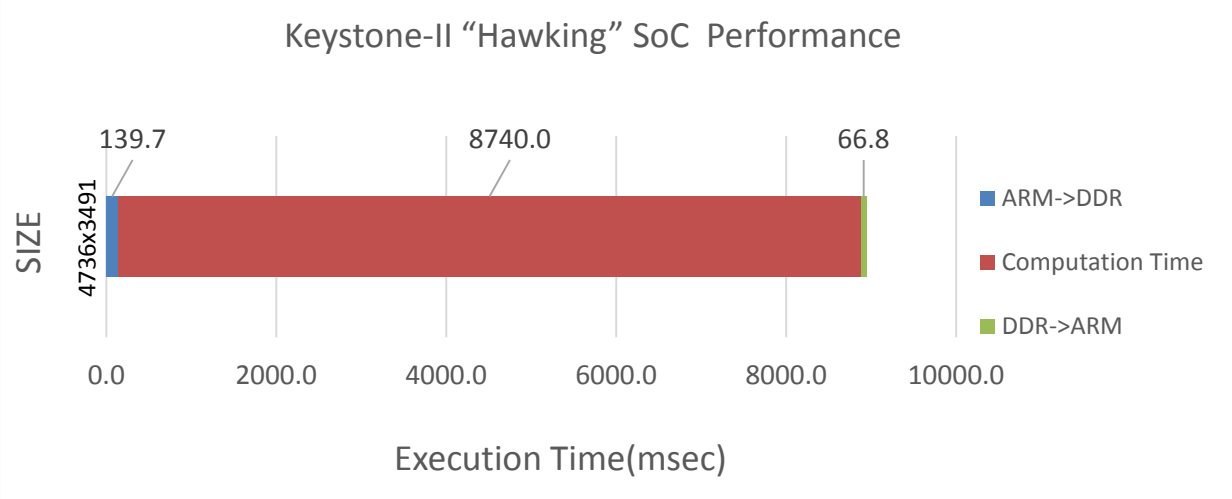

Figure 6.18 TI Keystone-II “Hawking” SoC Performance using DDR (4736x3491 image).

The above figures show that in both cases where either the MSMC memory or the DDR memory is used, the transfer time is significant. However, as it is already mentioned the memory copy

between the ARM core and the selected memory can be minimized using a more efficient memory accessing method. Another alternative which can be used for a better use of the bandwidth, would be a multi-threaded version of memory copy by splitting the amount of data to be copied between threads.

It has to be mentioned that for the parallelization of the specific algorithm the data parallelization method is used. The data are transferred from the ARM to the MSMC/DDR memory and each DSP unit works on a different part of the same data structure. For the 8-DSP configuration each DSP is responsible for the process of the 1/8 of the input image. For the specific application a row-wise division has been used. Moreover, due to the extra bounds that are needed for the creation of the necessary windows and patches a small part of the image is excluded from the process (indicated with arrows in Figure 6.19). Hence, if `SIZEY` declares the number of rows of the image, for the 8-DSP configuration, the image is divided in 8 chunks of $\frac{SIZEY-2*(R+F)}{8}$ rows each (Figure 6.19).

Figure 6.19 Row-wise image division of output

Another alternative would be to divide the image column-wise which is a more difficult and not so convenient procedure. Figure 6.20 presents an alternative which is used for applications where the input data affect the workload of each processor. For example if extra operations are needed to denoise a pixel and only the last part of the image (8th chunk) includes noisy pixels, then the workload of the 8th DSP is much higher than the workload of the other DSPs. Each colour corresponds to one of the 8 DSPs. Hence, by dividing the image in such a way is more luckily to divide the workload equally among the processors.

Figure 6.20 Row-wise image division of output – Optimal workload distribution

6.5.2. PDAK2H

An analytical performance model for the usage of the extra 16 DSPs (offered by the two extra "Shannon" SoCs of the Searay piggyback, Chapter 3) can be created. In this case the total execution time can be written as:

$$t_{exec}^{PU} = t_{comp}^{PU} + t_{trans}^{PU}$$

$$t_{exec}^{PU} = max \begin{cases} t_{comp}^{HWK} + t_{trans}^{HWK} \\ t_{comp}^{SHN0} + t_{trans}^{SHN0} \\ t_{comp}^{SHN1} + t_{trans}^{SHN1} \end{cases}$$

`HWK`, `SHN0` and `SHN1` are the three SoCs which offer the 24 in total (8 each) DSPs. Regarding the transfer time the following holds:

$$t_{trans}^{PU} = t_{trans}^{HWK} + t_{trans}^{SHN0} + t_{trans}^{SHN1}$$

It can be assumed that for an optimal work distribution:

$$t_{comp}^{HWK} + t_{trans}^{HWK} = t_{comp}^{SHN0} + t_{trans}^{SHN0} = t_{comp}^{SHN1} + t_{trans}^{SHN1}$$

Where, $t_{trans}^{HWK} < t_{trans}^{SHN0} = t_{trans}^{SHN1}$.

Moreover, we can write:

$$t_{trans}^{HWK} = t_{ARM \to HWK\_MEM} + t_{HWK\_MEM \to ARM} = \frac{N/3}{B_{ARM \to HWK\_MEM}} + \frac{N/3}{B_{HWK\_MEM \to ARM}}$$

$$t_{trans}^{SHN0} = t_{trans}^{SHN1} = t_{ARM \to SHN\_MEM} + t_{SHN\_MEM \to ARM} = \frac{N/3}{B_{ARM \to SHN\_MEM}} + \frac{N/3}{B_{SHN\_MEM \to ARM}}$$

The theoretical values of the above bandwidths are:
$B_{ARM \rightarrow HWK\_MSMC} = B_{HWK\_MSMC \rightarrow ARM} = 22.4GB/s$
$B_{ARM \rightarrow HWK\_DDR} = B_{HWK\_DDR \rightarrow ARM} = 12.8GB/s$

$B_{ARM \rightarrow SHN\_MSMC} = B_{SHN\_MSMC \rightarrow ARM} = 2GB/s$
$B_{ARM \rightarrow SHN\_DDR} = B_{SHN\_DDR \rightarrow ARM} = 2GB/s$

In the specific configuration the choice of the memory that is going to be used for the data storage is not made based on the complete amount of data. Only the 1/3 of the data need to be considered. Similar to the previous configuration it is possible to avoid the usage of the off-chip DDR memory by dividing the data to smaller chunks which fit in the MSMC memory of each SoC.

However, in the case of NL-M, the amount of data which are distributed to each SoC is greater than the 1/3 of the whole data. This happens because the windows and the patches that need to be created introduce some overlaps between the chunks. The division of the image is again row-wise. If `SIZEY` declares the number of rows of the image, for the 3-SoC configuration, the image is divided in 3 chunks of $\frac{SIZEY-2*(R+F)}{3} + 2 * (R + F)$ rows each. Hence the specific chunk includes $\frac{4}{3} * (R + F)$ more rows than the chunk which includes only the 1/3 of the whole data.

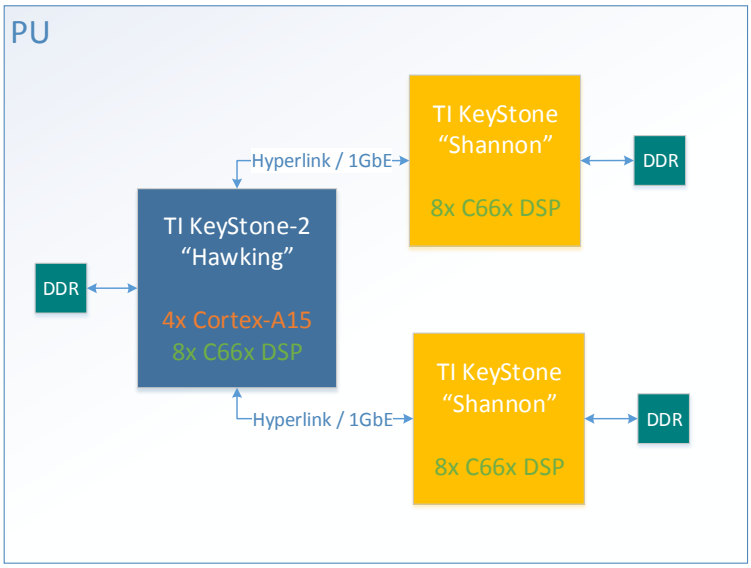

Figure 6.21 PDAK2H (PU) Model.

According to the aforementioned reasons, described in section 6.5.1, the transfer times between the host and the memories are not calculated based on the theoretical maximum bandwidths but the measured bandwidths are used instead. In addition to the reasons in section 6.5.1 one extra overhead is introduced because of the way that the data are transferred between the ARM core and the "Shannon" SoCs. The transfer is not completed immediately but it is divided into two steps. First the data are read from the DDR of the Hawking SoC (memory copy from the DDR to the MSMC) and then via DMA are transferred to the selected SoC. Table 6.15 shows the difference between the theoretical and the measured bandwidths from/to host to/from the Shannon MSMC/DDR memory.

	ARM->SHN_MSMC		SHN_MSMC ->ARM		ARM->SHN_DDR		SHN_DDR ->ARM	
	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured
Transfer Time (msec)	0.5	2.7	0.5	6.8	0.5	2.9	0.5	7
Bandwidth (GB/s)	2	0.4	2	0.1	2	0.4	2	0.1

Table 6.15 Theoretical and measured times of a 1MB memory copy between host (ARM of "Hawking") and "Shannon" MSMC/DDR memory.

6.5.2.1. Verification

Table 6.16 presents the comparison between the theoretical and the measured values of the execution time of NL-M in the 3-SoC configuration for four different sizes of input image. The execution time in the 3-SoC configuration includes the six memory transactions from and to the host (ARM) to and from the MSMC/DDR memory of each SoC (Figure 6.22, Figure 6.23 and Figure 6.24). The following formula indicates the selection between MSMC and DDR memory.

$$\text{MEM} = \begin{cases} \text{MSMC}, & (\text{SIZE} * \text{sizeof(float)} * 2)/3 \leq 6\text{MB} \\ \text{DDR}, & \text{else} \end{cases}$$

SIZE (memory)	512x512 (MSMC)		704x469 (MSMC)		1024x768 (DDR)		4736x3491 (DDR)	
Time (msec)	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured
t_{comp}^{DSP}*	1190		1270		6680		69880	
error (theoretical-measured)	-4%		+1%		-15.3%		-28.3%	
t_{exec}^{PU}	59.3	61.5	65.2	64.6	312.2	368.4	3625.9	5053.6
t_{comp}^{PU}	49.6	50.0	52.9	50.0	278.3	330	2911.7	3120.0
t_{trans}^{PU}	9.7	11.5	12.3	14.6	33.9	38.4	714.2	1933.6
$t_{\text{ARM}\rightarrow\text{HWK\_MEM}}$	0.9	0.9	1.1	1.1	2.8	2.5	57.3	46.6
$t_{\text{HWK\_MEM}\rightarrow\text{ARM}}$	2.2	2.1	2.8	2.6	10.5	11.2	220.5	218.7
$t_{\text{ARM}\rightarrow\text{SHN0\_MEM}}$	0.9	1.3	1.2	1.5	3.0	3.0	63.9	53.6
$t_{\text{SHN0\_MEM}\rightarrow\text{ARM}}$	2.4	2.4	3.0	3.0	7.3	7.4	154.3	781.2
$t_{\text{ARM}\rightarrow\text{SHN1\_MEM}}$	0.9	1.9	1.2	2.5	3.0	5.6	63.9	54.2
$t_{\text{SHN1\_MEM}\rightarrow\text{ARM}}$	2.4	2.9	3.0	3.9	7.3	8.7	154.3	779.3

*t_{comp}^{DSP} is the measured computation time in the single DSP configuration

Table 6.16 Theoretical and measured values of PDAK2H analytical performance model

Table 6.16 indicates that the difference between the theoretical and measured execution time is significant only for the biggest image (4736x3491). The computation time is predicted accurately (an acceptable 7% error) but the measured transfer time is almost three times bigger than the theoretical. This error occurs due to the underestimation of the transfer time from the “Shannon” DDR to the “Hawking” ARM. More specifically the theoretical transfer time from “Shannon” DDR to the “Hawking” ARM is predicted five times lower than the measured one. This happens because of two main reasons:

1. A single memory copy between the “Hawking” and the “Shannon” SoC cannot exceed the 4MB. Hence, multiple copies had to be used. The latter introduced a great overhead, especially during the reading procedure.
2. The DMA was not used for the data transfer because of some hardware issues that were noticed, but could not be fixed.

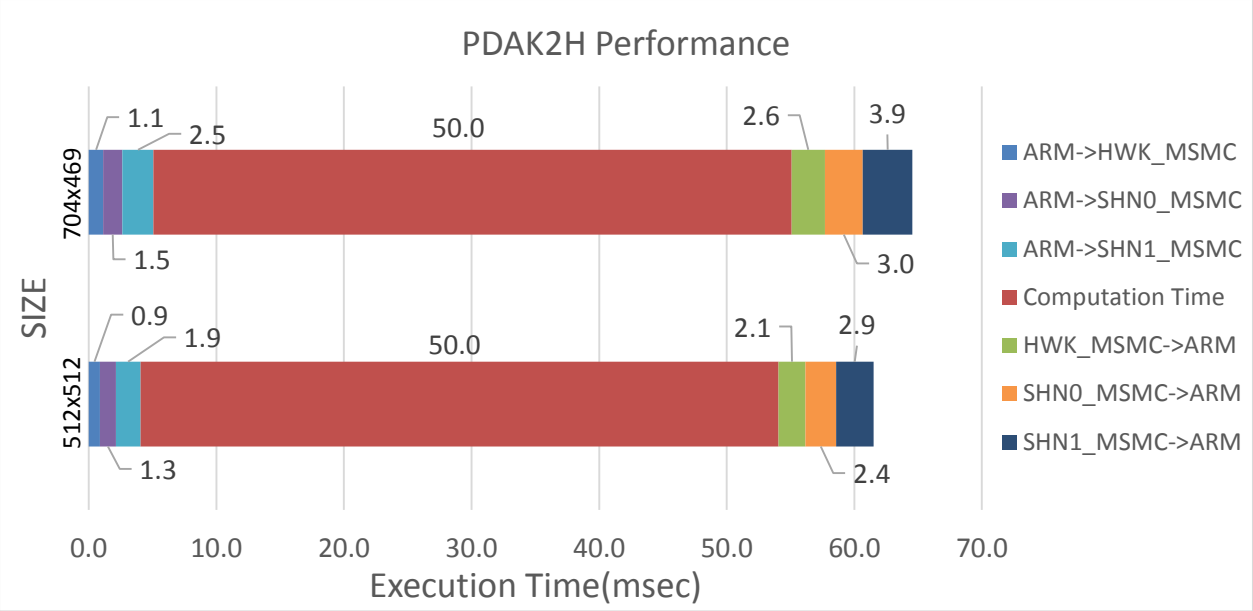

Figure 6.22 PDAK2H Performance using MSMC (512x512 and 704x469 images).

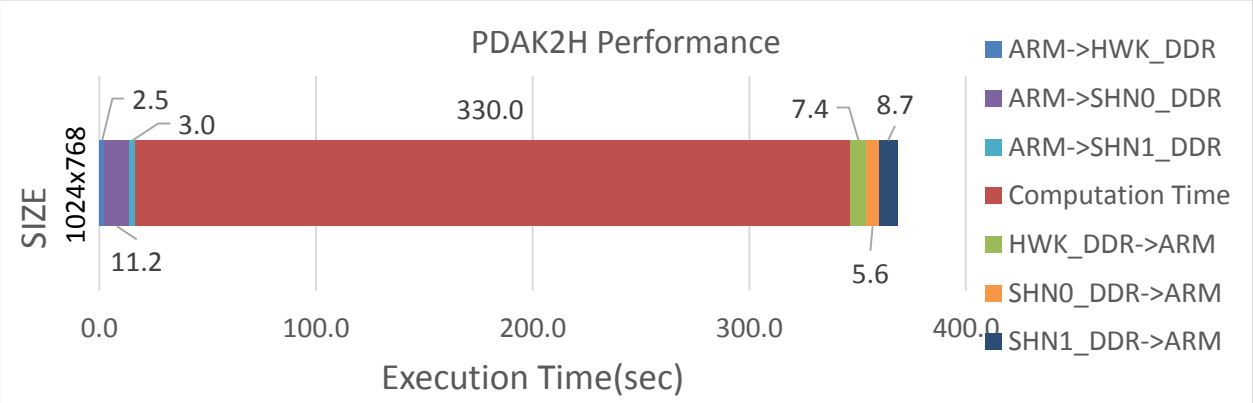

Figure 6.23 PDAK2H Performance using DDR (1024x768 image).

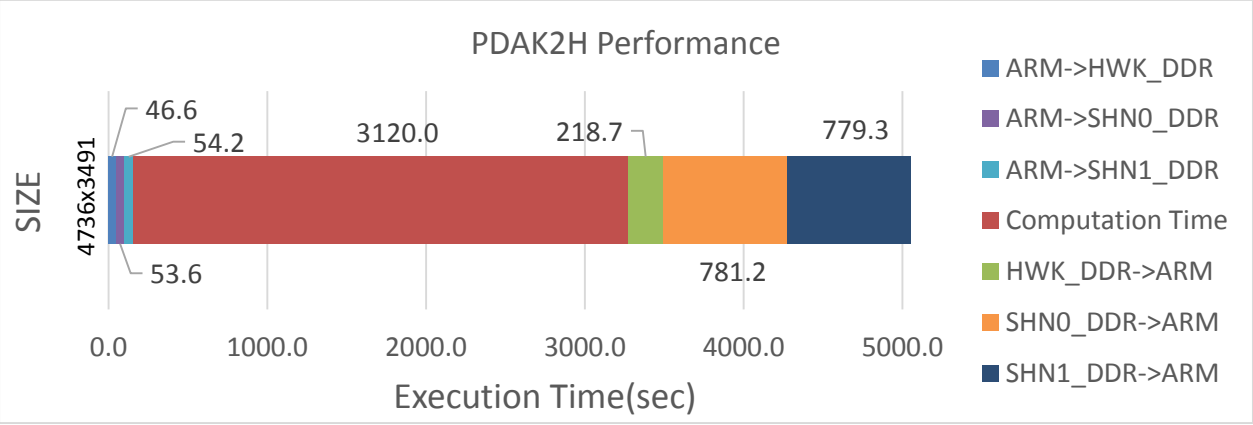

Figure 6.24 PDAK2H Performance using DDR (4746x3491 image).

According to the above figures, the transfer time in PDAK2H is larger than the previous one due to the two extra SoCs. However, the parallel distribution of the data to the available SoCs could speed up significantly the application. Moreover, as it is stated in the beginning of the subsection it could be feasible to divide the workload optimally between the available SoCs based on the additional time that is needed to transfer the data from the Hawking to the Shannon SoC.

It needs to be noticed, that for the data storage of the input image of size 1024x768, the DDR memory of the SoCs has been used instead of the MSMC memory. This was tried in order to measure the transfer times to/from DDR when a single memory copy is needed (in 4736x3491 image as was discussed a number of memory copies is needed to/from DDR).

6.5.3. ATCA-TK2-6PU Blade

ATCA-TK2-6PU blade is presented in section 3.2. It is an ATCA blade which is assembled with six identical PUs. Each PU contains: one TI Keystone-II (66AK2H14) and two Keystone-I (TMS320C6678) SoCs (see also Figure 3.7). The communication between a host x86 machine and the ATCA-TK2-6PU can be achieved via 10Gb Ethernet.

Regarding the analytical performance model of the ATCA-TK2-6PU blade, an extra level of hierarchy needs to be considered because of the introduction of a host (Figure 6.25) which is responsible for the distribution of the data among the six available PUs. Hence the transfer time in this model includes the transfer time form the host to the PUs and from the PUs to the host. The execution time of the ATCA-TK2-6PU blade is calculated as follows:

$$t_{exec}^{6PU\_BLADE} = t_{comp}^{6PU\_BLADE} + t_{trans}^{6PU\_BLADE}$$

$$t_{trans}^{6PU\_BLADE} = \frac{t_{trans}^{PU}}{6} + 6 * (t_{host\rightarrow PU} + t_{PU\rightarrow host}) = \frac{t_{trans}^{PU}}{6} + 6 * (\frac{N/6}{B_{host\rightarrow PU}} + \frac{N/6}{B_{PU\rightarrow host}})$$

The theoretical values of the bandwidth between the host and the PU is:
$B_{HOST\rightarrow PU} = B_{PU\rightarrow HOST}$ =1250MB/s

Similar to the previous configuration, the choice of the memory that is going to be used for the data storage is not made based on the complete amount of data. Only the 1/18 (eighteen available SoCs) of the data need to be considered.

Again, in the case of NL-M, the amount of data which are distributed to each PU is greater than the 1/6 of the whole data. As it is mentioned above, this happens because the windows and the patches that need to be created introduce some overlaps between the chunks. The division of the image is again row-wise. If `SIZEY` declares the number of rows of the image, for the 6-PU configuration, the image is divided in 6 chunks of $\frac{SIZEY-2*(R+F)}{6} + 2 * (R + F)$ rows each. Hence the specific chunk includes $\frac{10}{6} * (R + F)$ more rows than the chunk which includes only the 1/6 of the whole data.

Similar to sections 6.5.1 and 6.5.2, the bandwidth between the host and the PUs was measured. Table 6.17 shows the comparison between the theoretical and the measured bandwidth. A 10KB packet is transferred from/to the host to/from a PU via UDP sockets. It can be noticed that the measured bandwidth is much lower than the theoretical one. Especially in the PU→HOST case, the measured bandwidth is 25 times lower than the theoretical. However, according to Prodrive's benchmarking tests it is a hardware related issue which is still investigated.

	HOST->PU		PU->HOST	
	Theoretical	Measured	Theoretical	Measured
Transfer Time (msec)	0.01	0.02	0.01	0.2
Bandwidth (MB/s)	1250	500	1250	50

Table 6.17 Theoretical and measured times of a 10KB UDP packet transfer between host and a PU.

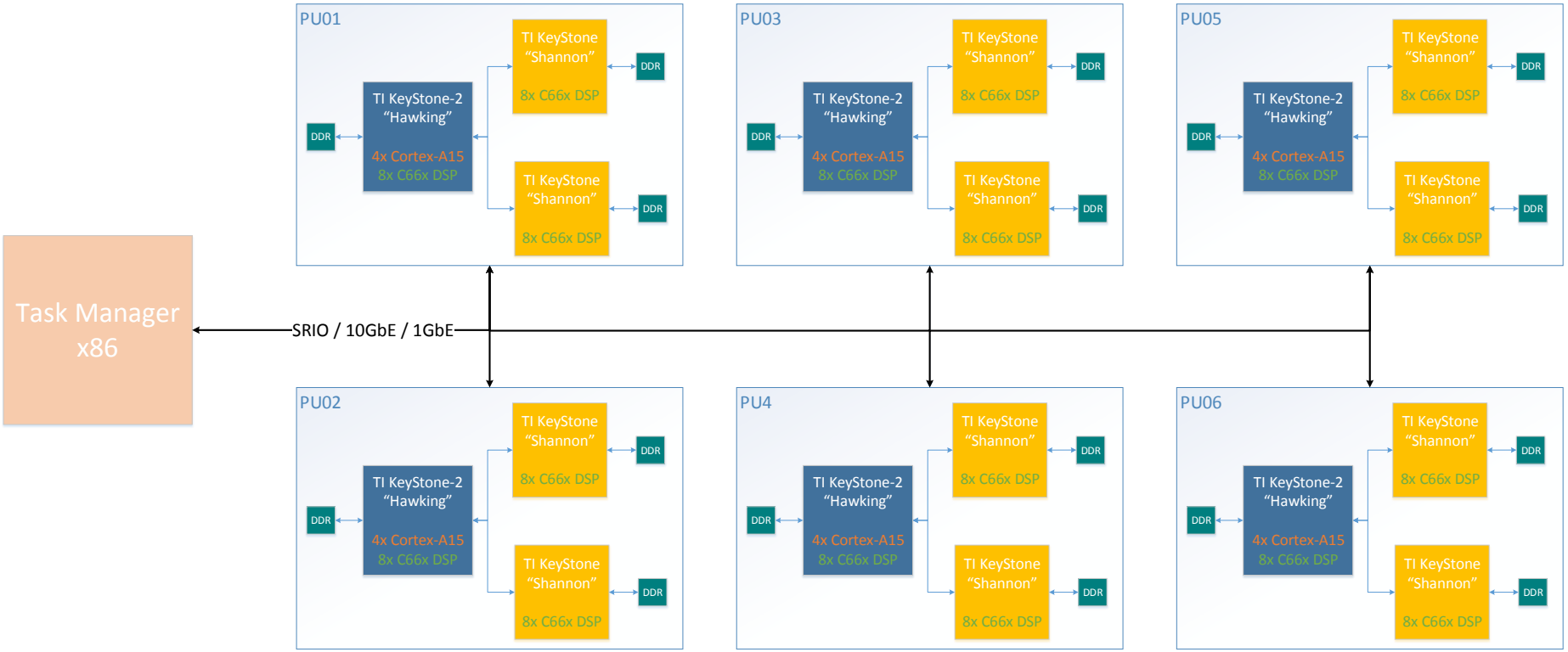

Figure 6.25 ATCA-TK2-6PU blade model.

6.5.3.1. Verification

Table 6.18 presents the comparison between the theoretical and the measured values of the execution time of NL-M in the 6-PU configuration for four different sizes of input image. The execution time in the 6-PU configuration includes twelve copies from/to the host (x86 machine) to/from each PU and six copies for each PU from and to the host (ARM) to and from the MSMC/DDR memory of each SoC. In Figure 6.26, Figure 6.27 and Figure 6.28 the six copies from the host to the PUs are notated as "*HOST->PUs"* and the six copies from the PUs to the host as "*PUs->HOST*". The following formula indicates the selection between MSMC and DDR memory.

$$\text{MEM} = \begin{cases} \text{MSMC}, & (\text{SIZE} * \text{sizeof(float)} * 2)/18 \leq 6\text{MB} \\ \text{DDR}, & \text{else} \end{cases}$$

SIZE (memory)	512x512 (MSMC)		704x469 (MSMC)		1024x768 (MSMC)		4736x3491 (DDR)	
	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured	Theoretical	Measured
t_{comp}^{DSP} *	1190		1270		6680		69880	
error (theoretical-measured)	-33%		-30%		-26%		-32%	
$t_{exec}^{6PU\_BLADE}$	33.5	49.7	39.7	57.0	120.9	163.3	2059.4	3037.8
$t_{comp}^{6PU\_BLADE}$	8.3	10.0	8.8	10.0	46.4	50.0	485.3	490.0
$t_{trans}^{6PU\_BLADE}$	25.2	39.7	30.9	47	74.5	113.3	1574.1	2547.8
$t_{\text{host}\rightarrow\text{PU}}$	0.4	0.3	0.4	0.3	1.1	2.4	22.0	34.2
$t_{\text{PU}\rightarrow\text{host}}$	3.5	5.8	4.4	7.0	10.5	15.4	220.5	327.7
$t_{\text{ARM}\rightarrow\text{HWK\_MEM}}$	0.2	0.2	0.2	0.3	0.4	0.5	9.6	8.8
$t_{\text{HWK\_MEM}\rightarrow\text{ARM}}$	0.4	0.5	0.5	0.4	1.1	1.0	36.7	38.0
$t_{\text{ARM}\rightarrow\text{SHN0\_MEM}}$	0.2	0.5	0.2	0.2	0.5	1.0	10.7	20.4
$t_{\text{SHN0\_MEM}\rightarrow\text{ARM}}$	0.4	0.4	0.5	0.5	1.2	1.0	25.7	145.7
$t_{\text{ARM}\rightarrow\text{SHN1\_MEM}}$	0.2	0.5	0.2	0.7	0.5	1.0	10.7	20.4
$t_{\text{SHN1\_MEM}\rightarrow\text{ARM}}$	0.4	1.0	0.5	1.1	1.2	2.0	25.7	143.1

*t_{comp}^{DSP} is the measured computation time in the single DSP configuration

Table 6.18 Theoretical and measured values of ATCA-TK2-6PU Blade analytical performance model.

Similar to the previous models the computation time is almost equal to the theoretical one. However there is a significant difference between the theoretical and the measured transfer times. As it is mentioned in 6.5.3 extra rows are needed to be transferred to each PU. These extra data are not included in the presented model. Moreover, the size of a UDP packet is limited. Hence, multiple packets need to be transferred for each PU. This procedure introduces an extra overhead. These two reason explain the difference between the theoretical and the measured transfer time.

From Figures Figure 6.26, Figure 6.27 and Figure 6.28 it is made clear that the dominant part of the execution time in the 6PU configuration is the time needed to transfer the output data from the PUs to the host. Specifically, for the 4736x3491 image, if the bandwidth of a memory copy from the PUs to the host was equal to the bandwidth from the host to the PUs ($B_{HOST\rightarrow PU} = B_{PU\rightarrow HOST}$) then the achieved speedup over the single core configuration would be more than two times greater than the one that it is achieved now (23x, see also Figure 6.29). Hence, the focus of a future research needs to be turned on the minimization of the specific transfer time.

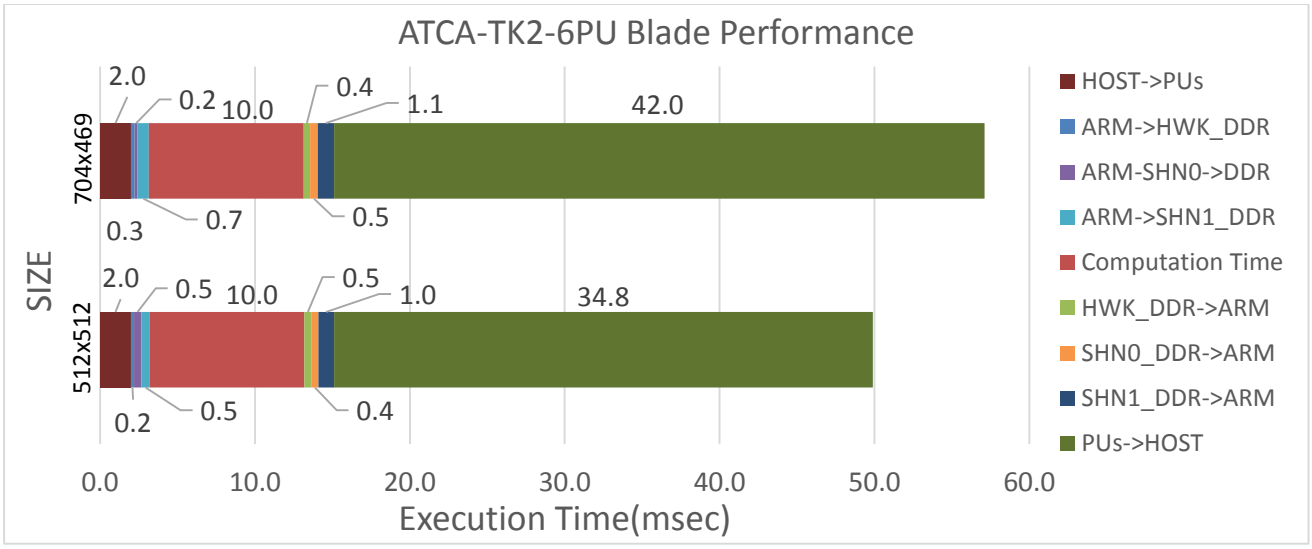

Figure 6.26 ATCA-TK2-6PU blade performance using MSMC (512x512 and 704x469 images).

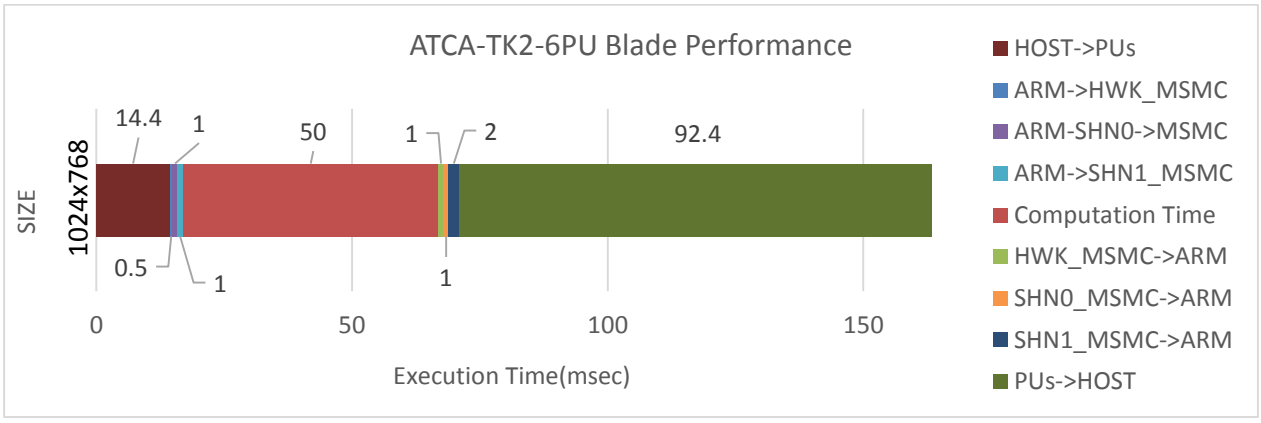

Figure 6.27 ATCA-TK2-6PU blade performance using MSMC (1024x768 image).

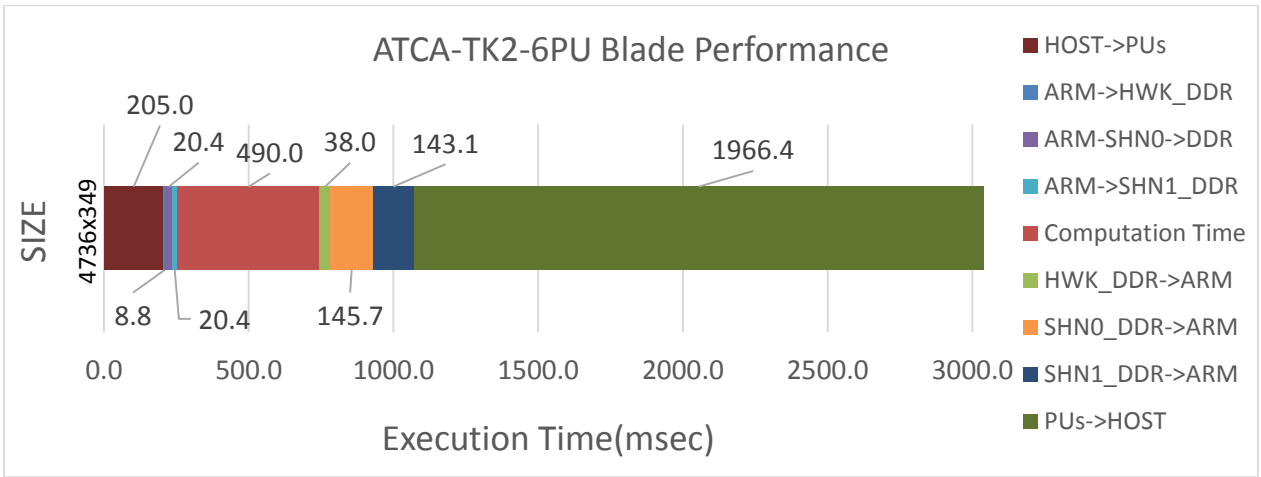

Figure 6.28 ATCA-TK2-6PU blade performance using DDR (4736x3491 image).

Figure 6.29 presents the comparison between the perfect, the theoretical and the measured speedup, which was achieved for the 4736x3491 image, over the single core configuration. It is clear that the time needed for the data transaction among the available processing units affect significantly the achieved speedup and must be taken into account before the selection of a target platform. Moreover, Figure 6.29 shows that the analytical performance models described in

sections 6.5.1, 6.5.2 and 6.5.3 give a much better estimate of the achieved performance than the formulas which take only the computation time into account (section 6.5).

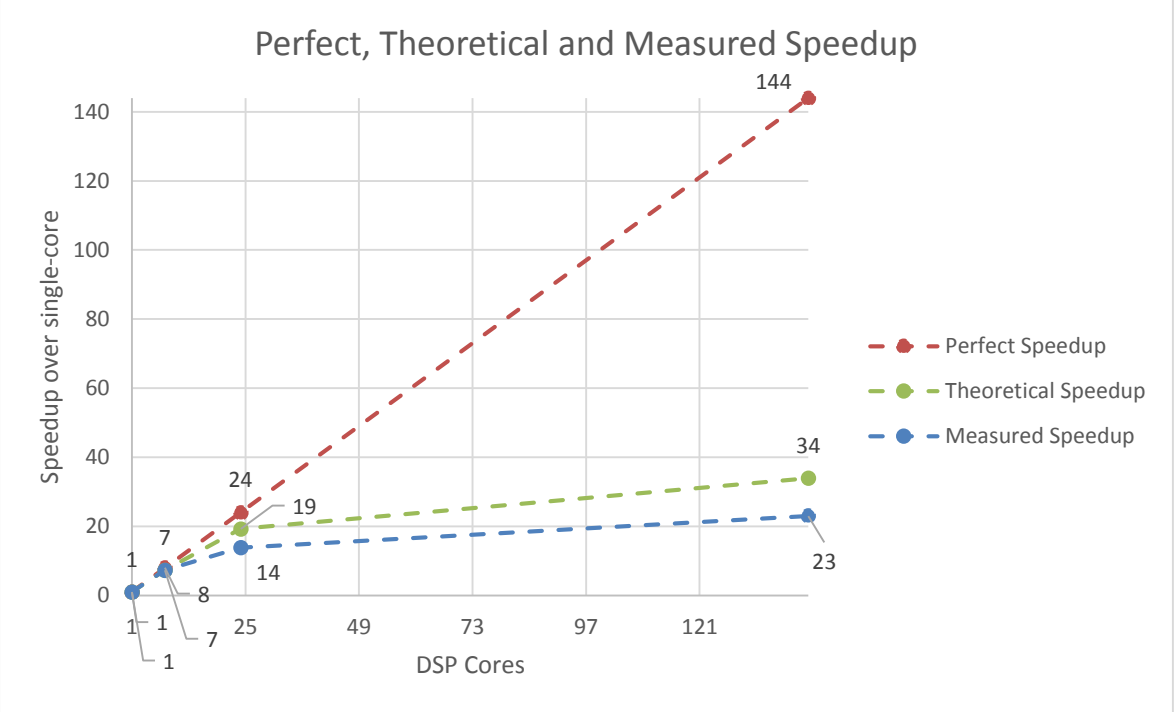

Figure 6.29 Perfect, theoretical and measured speedup over the single-core configuration for 4736x3491 image in NL-M.

7. Analysis

In the following subsections a set of guidelines for selecting a target platform based on an application's characteristics are introduced. Also a methodology to distribute optimally an application over a heterogeneous platform is given.

7.1. Target selection based on application's characteristics

It was shown that the time that is needed to transfer data among the available processing units affects immediately the execution time of a distributed application. Hence, in addition to the computation time, the transfer time needs to be taken into account also.

The transfer time is depended on both application and platform characteristics. Regarding the application characteristics, the amount of data that needs to be transferred to each processing unit has to be considered. In Chapter 6, the analytical performance models which have been constructed assume that the input data will be distributed equally among the available processing units. However, this is not always the case. Applications can be divided into two main categories:

A. Applications which does not need additional knowledge (no extra data are needed) and
B. Applications which need additional knowledge (extra data are needed)

A. In applications like vector addition, processing a portion of data does not require the use of additional amounts of data. Hence, according to the analytical performance models described in section 6.5 the input data are going to be divided equally among the available processing units. Hence, the transfer time for a distributed platform which is consisted by p processing units, can be calculated as follows:

$$t_{trans}^{p\_pu} = \frac{N/p}{B_{host \to pu}} * p + \frac{N/p}{B_{pu \to host}} * p$$

N: Total amount of data
$B_{host \to pu}$: Bandwidth from host to processing unit
$B_{pu \to host}$: Bandwidth from processing unit to host

Which is equal with the time that is needed to transfer the necessary data to a single processing unit:

$$t_{trans}^{p\_pu} = t_{trans}^{single\_pu} = \frac{N}{B_{host \to pu}} + \frac{N}{B_{pu \to host}}$$

In the above it is assumed that the bandwidth is equal between the host and all the available processing units.

In the above, it was assumed that the size of the input and the output data is equal:

$$N_{in} = N_{out} = N$$

Moreover, if the distributed platform allows it, the parallelization of the data distribution will decrease the transfer time proportionally to the number of the available cores or threads which are responsible for the distribution:

$$t_{trans}^{p\_pu} = \frac{\frac{N/p}{B_{host \to pu}}}{d} + \frac{\frac{N/p}{B_{pu \to host}}}{d} = \frac{t_{trans}^{single\_pu}}{d}$$

Where, d is the number of the cores or threads which are responsible for the data distribution.

In that case, the total execution time can be written as the sum of the computation and the transfer time:

$$t_{exec}^{p\_pu} = t_{trans}^{p\_pu} + t_{comp}^{p\_pu} = \frac{t_{trans}^{single\_pu}}{d} + \frac{t_{comp}^{single\_pu}}{p}$$

And the following holds:

$$t_{exec}^{p\_pu} \leq t_{exec}^{single\_pu}$$

The equality holds only if $d = p = 1$.

B. On the other hand, in applications like NL-M each, a processing unit needs additional data in order to process the portion which correspond to them. Hence, the transfer time for a distributed platform which is consisted by p processing units, can be calculated by:

$$t_{trans}^{n\_pu} = (\frac{N/p}{B_{host\rightarrow pu}} + \frac{N_{extra}}{B_{host\rightarrow pu}}) * p + (\frac{N/p}{B_{pu\rightarrow host}} + \frac{N_{extra}}{B_{pu\rightarrow host}}) * p = t_{trans}^{single\_pu} + p * (\frac{N_{extra}}{B_{host\rightarrow pu}} + \frac{N_{extra}}{B_{pu\rightarrow host}})$$

N: Total amount of data
N_{extra}: The additional amount of data which each processing unit needs
$B_{host\rightarrow pu}$: Bandwidth from host to processing unit
$B_{pu\rightarrow host}$: Bandwidth from processing unit to host

In that case the time that is needed to transfer the necessary data to a single processing unit increases by $p * (\frac{N_{extra}}{B_{host\rightarrow pu}} + \frac{N_{extra}}{B_{pu\rightarrow host}})$.
Once again the data distribution can be parallelized and the transfer time will be decreased as follows:

$$t_{trans}^{n\_pu} = \frac{t_{trans}^{single\_pu}}{d} + \frac{p * \left(\frac{N_{extra}}{B_{host\rightarrow pu}} + \frac{N_{extra}}{B_{pu\rightarrow host}}\right)}{d}$$

Now, the total execution time can be written as:

$$t_{exec}^{n\_pu} = t_{trans}^{n\_pu} + t_{comp}^{n\_pu} = \frac{t_{trans}^{single\_pu}}{d} + \frac{p * \left(\frac{N_{extra}}{B_{host\rightarrow pu}} + \frac{N_{extra}}{B_{pu\rightarrow host}}\right)}{d} + \frac{t_{comp}^{single\_pu}}{p}$$

Figure 7.1 presents the speedup over a single core configuration with respect to the ratio between the computation and transfer time of the single core configuration. In the considered configurations the platform consists of 8, 16 and 24 processing units and 4 distributors. Moreover, 10% of the total amount of data are considered as additional data that needs to be transferred to each processing unit. The bandwidth between the distributors and the processing units is set to 2GB/s.

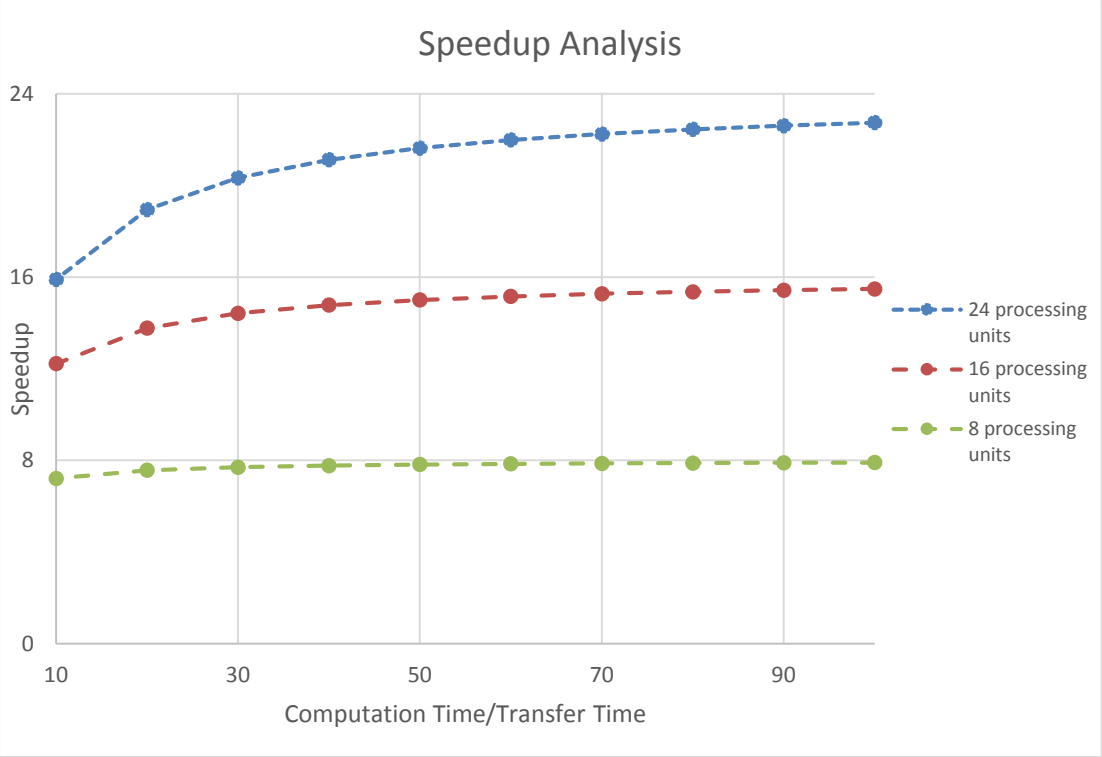

Figure 7.1 Speedup affected by the ratio between computation and transfer time

According to the above theoretical analysis and the specific configurations the only platform that achieves the maximum theoretical speedup is the one with the 8 processing units. The latter indicates that the speedup which is gained by the decrease of the computation time is cancelled by the increase of the data transfer time among the processing units.

Moreover, Figure 7.2 indicates the behaviour of the above configurations when the computation time of the single core configuration is 100 times greater than the transfer time. The speedup varies with respect to the ratio between the additional data that are needed to be transferred among the processing units and the total amount of data.

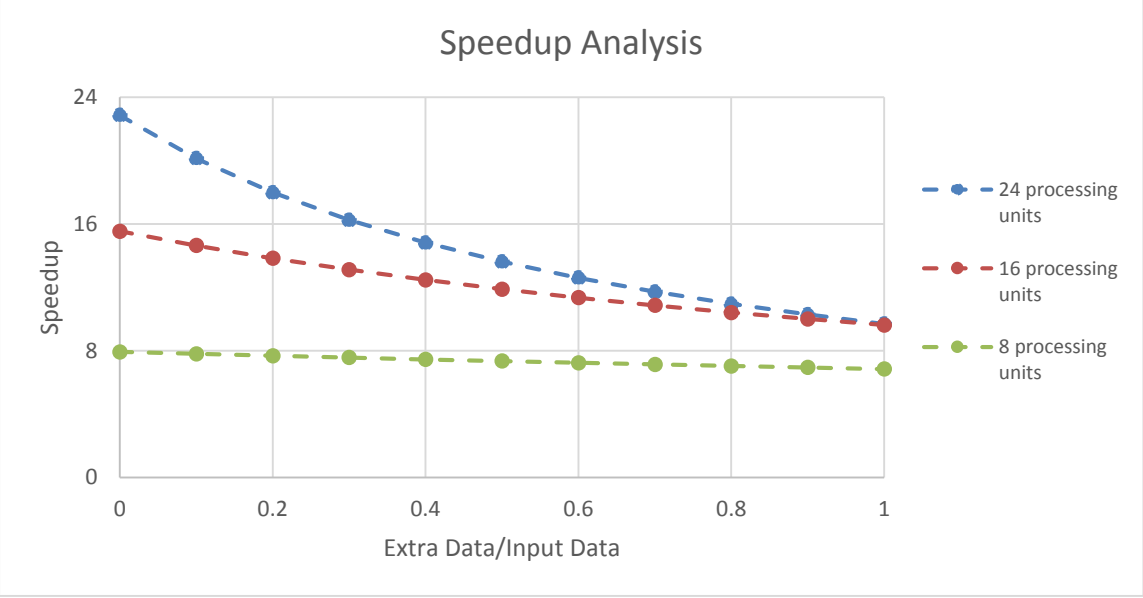

Figure 7.2 Speedup affected by the ratio between extra and input data

Similar to the previous discussion, the speedup which is gained by the additional processing units is cancelled by the time that is needed to transfer the necessary data among the processing units. However, the configuration which is consisted by 8 processing units is not affected significantly by the increase of the extra transactions.

Both examples indicated the importance of selecting the suitable target platform based on the characteristics of the application. This is even more important when there is a need for continuous transactions between the processing units. A more complex analytical performance model would be needed to describe such a case.

7.2. Optimal distribution of an application

Several design choices have been made or discussed during this thesis research attempt. The majority of them concerned mainly the distribution of an application over different distributed platforms. The most general of the considered choices are discussed in this subsection.

7.2.1. Division and distribution of the input data

The first step in every distributed application is the division and the distribution of the data among the available processing units. In subsection 6.5 an analysis is made regarding the choices about the data division. If the input data affect the workload of a processing unit then a more careful division is advised. The method, which was used to divide the input data during experimentation, does not take into account the characteristics of the input data. It divides row-wise the input data to as many chunks as the number of available processing units. Figure 7.3 represents the division of an input image when the target platform is the ATCA-TK2-6PU blade. First, the input image is divided to 6 equal parts, and each part is distributed to one the processing units. Then, each processing unit divides its part to 3 equal parts, and each part is distributed to one of the SoCs, in order to be processed by the 8 available DSPs.

Figure 7.3 Row-wise division and distribution in the ATCA-TK2-6PU Blade

In most cases it is preferred to perform less transactions with more data than more transactions with less data. Extra transactions between the processing units will introduce additional overhead. However, it is also common to divide the data in smaller chunks in order to fit in smaller, on-chip, faster memories.

7.2.2. Workload distribution

The next step is to select the processing units which will be responsible for the process of the data. Every available processing using has a task and a role during the execution of the application. The roles and the tasks must be assigned to the processing units based on their characteristics. For instance, in subsection 6.3.3 it is shown that the performance of a processing unit affects the optimal workload distribution and the performance of the application. The ARM cores in that case were responsible for a part of the process. However, in subsection 7.1 it is showed that the number of available distributors affect the performance of the applications.

7.2.3. Programming Method

The developer is not always able to parallelize the application manually. As it is made clear from the previous discussion both the process of parallelization and the distribution of the data are really time consuming. In that case several options exist in order to automate the parallelization of the application. For the investigated platforms two automated programming methods have been

examined, OpenCL and OpenMP. However, only the single Hawking SoC can be programmed via the two aforementioned programming paradigms. Hence, if the 16 additional DSPs, offered by the PDAK2H, are needed the “plain-C” method has to be selected. Extra DSPs can be used via OpenCL or OpenMP if two or more PDAK2Hs are combined. Either one of the PDAK2Hs has to be used as a manager in order to divide and distribute the data or a host-manager machine needs to be used. On the other hand it is showed that the choice of the programming method is highly dependant on the characteristics of the application. If there is a need of mapping functions or header files to the target device, the current version of OpenMP introduces a great overhead which is not allowed to real time applications. On the other hand in OpenCL a significant duration of time is spent for the initialization of the device. However, a stream-line application would not be affected by such a delay.

However, by using a source-to-source transformation tool as “Bones” the developer is able to extract automatically efficient and readable parallel code for parallel architectures (OpenMP, OpenCL or CUDA). Based on the investigation and the experiments that were performed during this thesis regarding OpenCL and OpenMP (section 6.4), it was shown that both programming paradigms can be used efficiently in the TI Keystone-II “Hawking” SoC. Hence, a backend for the “Bones” tool would help the developer by providing OpenCL or OpenMP code for the target heterogeneous SoC.

Moreover, a source-to-source code transformation generating host and device code from the initial OpenMP/OpenCL single device code could be used for multiple TI Keystone-II “Hawking” SoCs or for the ATCA-TK2-6PU Blade. The host code would be responsible for:

1. Initializing the device
2. Dividing and the distributing the input data among the available devices
3. Triggering the execution of the device code
4. Receiving the data from the devices

The initial OpenMP/OpenCL single device code can be generated via the new backend for the “Bones” tool.

Details about the “Bones” tool are given in Appendix A.2.

8. Conclusions

As the number of cores and SoCs in a platform increases multiple issues, like scalability issues or communication issues (latency, overhead, etc.), must be taken into account. Additional issues, like non-uniformity or differences between the available programming methods, arise when multiple types of processors are combined in the same SoC. Hence, selecting the most suitable target platform for a specific application is not a simple procedure. However, analytical performance models can be used in order to give an accurate prediction about the capabilities and the performance of the target system.

In this document, analytical performance models for DSP heterogeneous platforms were deducted. Two platforms developed by Prodrive-Technologies were selected to be used as the target hardware throughout all experiments in this report.

1. PDAK2H: a computer mezzanine card which is assembled with the TI Keystone-II "Hawking" SoC which offers in total 4 A15 ARM cores and 8 C66x DSP cores. The introduction of two additional SoCs (TI Keystone-I "Shannon" SoC), offering in total 16 extra DSPs, on PDAK2H is feasible.
2. ATCA-TK2-6PU blade which consists of 6 Processing Units (PUs) containing per PU: one TI Keystone-II "Hawking" and two Keystone-I "Shannon" SoCs offering 4 A15 ARM cores and 24 C66x DSP cores in total per PU.

Both types of processors (ARM A15 and DSP C66x) of the TI Keystone-II "Hawking" SoC have been benchmarked in detail by a series of microbenchmarking applications. More specifically, it has been shown that the ARM A15 maximum performance is reachable only when the SIMD instructions are used. The performance of 42 GFLOPs (88% of the theoretical maximum) was achieved by the development of an application where the ARM NEON C intrinsics were used. In applications were the SIMD instructions are not used only the 1/4 of the maximum performance is reachable. Regarding the verification of the theoretical maximum performance of the C66x core, a LINPACK micro-benchmarking application has been tested. 32 GFLOPS (Double Precision) were achieved. DMAs need to be used in order to achieve the maximum bandwidth between both A15 and C66x cores and the MSMC shared on-chip memory.

For demonstrating and testing the capabilities of the TI Keystone-II "Hawking" SoC a compute intensive, image denoising algorithm called Non Local Means (NL-M) was selected to be ported to this architecture. The behaviour of NL-M algorithm has been analysed in the constructed roofline models for the TI Keystone-II "Hawking" SoC. Several modifications haven been made on the source code of NL-M. By replacing the `expf()` function by its approximation using a Taylor series a great speedup (40x in some cases) was achieved. The additional advantage of this replacement is the more accurate counting of the performed floating point operations. Finally, a performance of more than 50Gflops has been achieved, which is more than 30% of the theoretical maximum performance. A significant performance penalty is induced otherwise.

Also, a comparison between the "plain-C", OpenMP and OpenCL programming methods has been made. First results showed OpenCL deployment performs better than OpenMP. "plain-C" and OpenCL shows almost the same behaviour. However, OpenCL includes a great overhead at the start time due to initializations that need to be done. It was shown that the OpenMP implementation includes also a significant overhead because of the inclusion of several header files and functions in the target application. "plain-C" was selected to be used during the next steps of the assignment due to the aforementioned overheads of the other two programming methods. Also, the 16 extra DSPs of the PDAK2H can be programmed only via the "plain-C", since the necessary libraries are not available.

By performing a series of experiments, the scalability issues of the TI Keystone-II "Hawking" SoC have been examined. In homogeneous configurations, for both ARM and DSP cores, a proportional to the number of available processors decrease in the computation time has been observed. For the 8-DSP configuration a 7x speedup over the single-DSP configuration is achieved. In the single core experiments the difference between the execution time in the ARM and the DSP processor was small, ARM was only 1.2 times slower than DSP. During the experiments where both types of processors have been involved, the optimal workload allocation among the processors has been

calculated. In this case, for a 1024x768 image, a significant decrease from 11.4 seconds (8-DSP configuration) to 7.6 seconds (4-ARM + 8-DSP configuration) has been achieved, a 30% performance gain. However, for a real-time application (i.e. video-stream) the above time has to be even lower in order to achieve i.e. 30fps. Hence, the introduction of extra DSP cores is investigated.

The PDAK2H and the ATCA-TK2-6PU blade which offer in total 24 and 144 DSPs respectively have been examined. In both cases the achieved speedup over the single core configuration was much lower than the perfect speedup. Specifically, for a 15MPixel image the achieved speedup was:

- 14x (almost two times lower than the perfect speedup) in the PDAK2H which contains 24DSPs and
- 23x (more than six times lower than the perfect speedup) in the ATCA-TK2-6PU Blade which contains 144DSPs

Experiments showed that the main reason of the difference between achieved and perfect speedup is the time that is needed to transfer the data to/from the available processing units.

Analytical performance models have been constructed for three different heterogeneous DSP architectures:

1. TI Keystone-II “Hawking” SoC (8 DSPs)
2. PDAK2H (24 DSPs)
3. ATCA-TK2-6PU Blade (144 DSPs)

It was shown that in addition to the computation time (t_{comp}), the time needed to transfer the data among the available processing units (t_{trans}) must be taken into account. Hence, the execution time of an application (t_{exec}) can be written as the addition of the computation time and the transfer time:

$$t_{exec} = t_{comp} + t_{trans}.$$

For each analytical performance model the measured computation time is almost equal to the theoretical one. This means that for every examined platform, the ideal speedup, regarding the computation time, is achieved. Regarding the transfer time, experiments have shown that the theoretical bandwidths between the host and the available processing units are not achieved for several reasons. Thus, for every model the achievable bandwidths have been measured.

However, even if the measured achievable bandwidths were used in the models, in some cases a difference was noticed between the measured and the theoretical execution time in the NL-M application.

For a 15MPixel image, in the TI Keystone-II “Hawking” SoC the measured execution time was almost equal with the theoretical execution time. The error of the model was only 0.2%.

In the PDAK2H, the execution time, for the same image, was underestimated by almost 28%. This error occurs because:

1. A single memory copy between the “Hawking” and the “Shannon” SoC cannot exceed the 4MB. Hence, multiple copies had to be used. The latter introduced a great overhead, especially during the reading procedure.
2. The DMA was not used for the data transfer because of some hardware issues that were noticed, but could not be fixed.

In the ATCA-TK2-6PU blade the execution time for the process of a 15MPixel image was underestimated by 33%. It was shown that this difference occurs mainly due to the time needed to transfer the output data from the PUs to the host. Hence, the focus of a future research needs to be turned on the minimization of this specific transfer time. If the bandwidth of a memory copy from the PUs to the host was equal to the bandwidth from the host to the PUs ($B_{HOST \rightarrow PU} = B_{PU \rightarrow HOST}$) then the achieved speedup over the single core configuration would be more than two times larger than the one that it is achieved now.

Finally, alternative designs, such as the usage of the DMA or the parallel distribution of the data, have been proposed as future work. These methods can improve the application by decreasing the transfer time among the processing units, by using the bandwidth to/from the processing units in a more efficient way.

It was shown that the discussed analytical performance models can give a much more accurate estimation, regarding the execution time, than the formulas which take into account only the computation time. In this way, a user can buy and test his application in a simple cheap evaluation module which is composed by a single DSP and via the analytical performance models determine the extra DSPs, SoCs or PUs which are needed for his application.

Finally, a benchmarking tool has been developed in order to benchmark and demonstrate the capabilities of processing units whose main component is the TI Keystone-II “Hawking” SoC. Two examples of such processing units are the PDAK2H and the ATCA-TK2-6PU blade which were employed throughout all experiments in this report.

Appendix A. Further research aspects

In the following chapter firstly the feasibility of employing the OpenMPI library over RapidIO is going to be examined and secondly an investigation is made regarding the development of a backend for Bones tools for the mutli-SoC heterogeneous platforms.

A.1. OpenMPI on RapidIO

RapidIO is a high-performance packet-switched fabric interconnect standard designed for use in embedded systems. OpenMPI is a Message Passing Interface (MPI) library. A merger of LAM/MPI, LA-MPI and FT-MPI projects.

A.1.1. RapidIO Technical Overview

RapidIO provides chip-to-chip, board-to-board and shelf-to-shelf peer to peer connectivity. Its development was driven by the following initial design goals:

- Focus on embedded control plane applications
- Scope limited to in-the-box or chassis applications in the embedded space
- Lower cost and pin count
- Limited software impact
- Simple switches
- Protocol extensibility

A.1.2. Protocol Overview

RapidIO [12] [13] uses a three-layer architectural hierarchy (Figure A.1). The highest layer is the logical layer which is responsible for the necessary information for endpoints to initiate and complete transactions. The transaction layer follows, which defines how logical layer request transactions are routed from end point to end point. The lowest layer is the physical layer. It specifies the link protocol, the electrical characteristics of the physical link and a low-level error management.

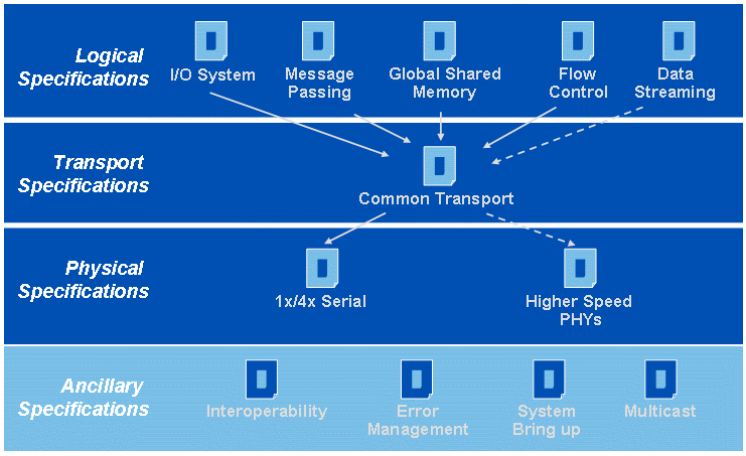

Figure A.1 RapidIO layer hierarchy [13]

A.1.3. Packet Format

Providing a message, based interface, it is capable of speeds up to 8.25 Gb/s (gen2) and 10 Gb/s (gen3) full duplex per lane. Messages are the communication element between end point devices in the system. Figure A.2 illustrates the transaction progress through a RapidIO system where a master (initiator) generates a request transaction, which is transmitted to a target with the help of a number of switched devices (fabric). The operation is completed after the response transaction from the target to the initiator.

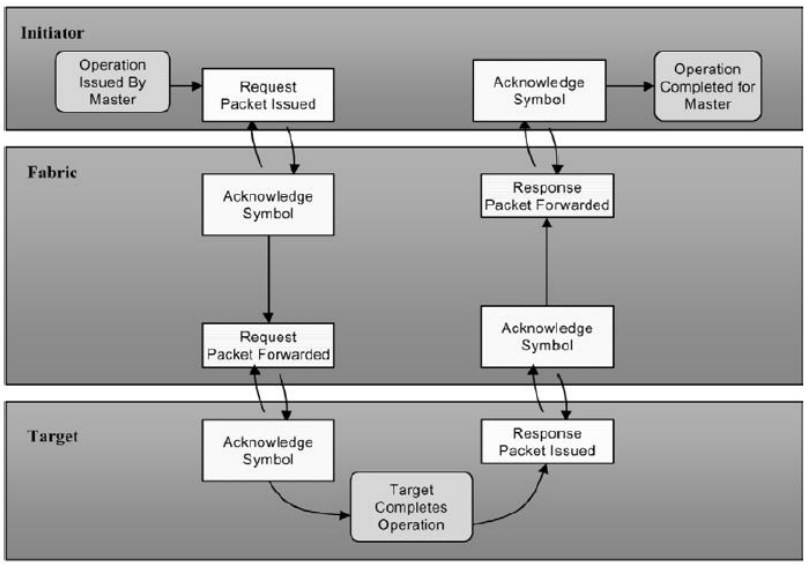

Figure A.2 Transaction progress through a RapidIO fabric [13]

Figure A.3 shows the format of a RapidIO message request packet. The maximum size of a packet in a RapidIO fabric is 256 bytes. Messages larger than 256 bytes are segmented into multiple messages and reassembled at the destination even if they arrive out of order. Moreover, a single transaction between a source and destination cannot exceed the 16 multi-segment messages.

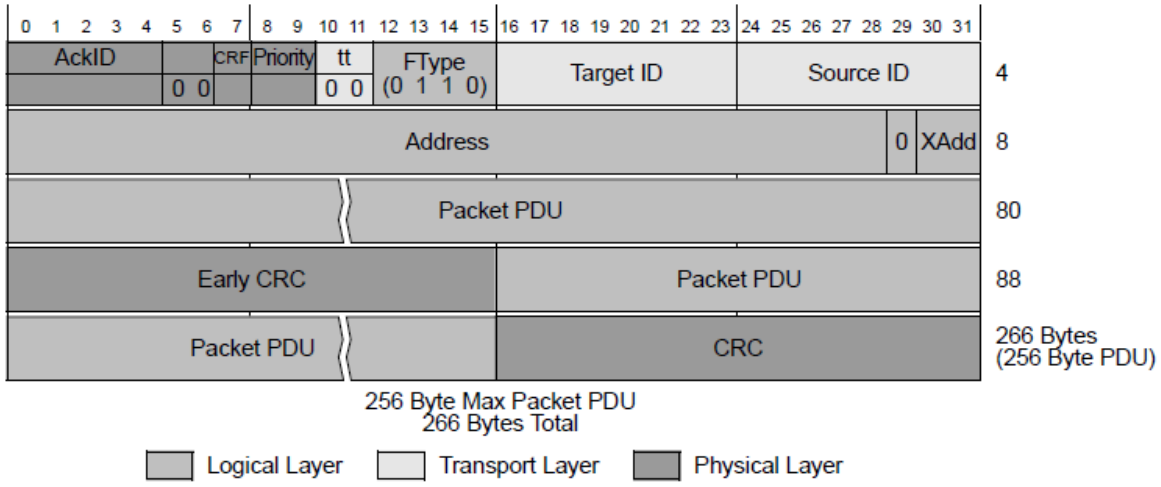

Figure A.3 RapidIO Packet Format [13]

A.1.4. Globally Shared Memory

A RapidIO fabric is able to support a globally shared distributed memory system. This happens due to the fact that RapidIO defines cache coherence and OS support operations that allow a processing device with a cache to keep its cache state coherent with others in the fabric. More specifically, a directory-based coherency system is specified where each memory controller is responsible for tracking where the most current copy of each data element resides in the system.

A.1.5. Flow Control and Deadlock Avoidance

Regarding the congestion that can be exhibited in a fabric, RapidIO supports a flow control facility in the logical layer which can decide whether it should shut off the traffic from a specific source endpoint.

The second category of control flow mechanisms is specified in physical layer where three types of mechanisms are presented.

- Retry mechanism: The receiver has the ability to send a retry control symbol to the sender if the received packet is corrupted or if there is a lack of resources.
- Throttle mechanism: The specific mechanism is used whenever it is necessary to slow down a transaction. The idle control symbol is used.
- Credit-based mechanism: With this mechanism the transmitter knows whether to transmit a packet based on receiver's buffer status.

Moreover, by prioritizing the requested flows and by associating ordering rules it is able to avoid dependency loops and deadlocks. Regarding the distinction between requests and responses, a higher priority is assigned to responses than the associated request. Using the above rules the following table is determined.

Overall Physical Priority PRIO field	Flow A	Flow B	Flow C
3	Response	Response	Response
2	Response	Response	Request
1	Response	Request	
0	Request		

Table A.1 Two-Bit Physical Priority and Logical Flows

A.1.6. Linux RapidIO Subsystem

According to [14], RapidIO subsystem follows the standard Linux Device Model in a similar way to the other buses in the kernel. Its core consists of four major components:

1. Master Port: a bridge between the processor which runs the Linux code and the switched fabric network. It is responsible for the generation and the reception of RapidIO packets and it is represented by a *rio_mport* data structure.
2. Device: an endpoint (not a master port) or a switch on the network which is represented by a *rio_dev* data structure.
3. Switch: a special class of device which is used to route packets between point to point connections towards their final destination. Its *rio_dev* data structure is expanded by *rio_switch* data structure, which consists of switch necessary information.
4. Network: a set of endpoints and switches that are interconnected. It is represented by a *rio_net* data structure which contains all the necessary information about the RapidIO network.

A.1.7. OpenMPI

Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function a wide variety of parallel computers. It includes a great number of library routines used in portable message-passing programs in different languages such as Fortran, C, C++ and Java. OpenMPI can be described as a merger of three different MPI implementations: LAM/MPI, LA-MPI and FT-MPI. However, in [15] it is stated that "OpenMPI is an all-new implementation of the MPI", since "it provides functionality that has not previously been available in any single, production-quality MPI implementation".

A.1.8. OpenMPI architecture

OpenMPI consists of three main functional areas (Figure A.4):

1. Modular Component Architecture (MCA): a component-based architecture that provides management services for all other layers.
2. Component frameworks: the necessary back-end component framework which manages a number of modules.
3. Modules: self-contained software units that export well-defined interfaces.

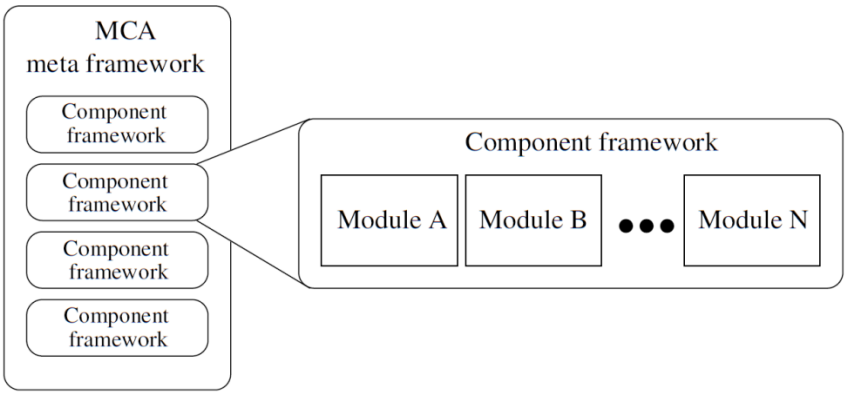

Figure A.4 OpenMPI functional areas [15]

Moreover, OpenMPI can be divided in three main abstraction project layers (Figure A.5):

1. Open Portable Access Layer (OPAL): Open MPI's core portability between different operating systems and basic utilities.
2. Open MPI Run-Time Environment (ORTE): Launch, monitor individual processes, and group individual processes in to “jobs”
3. Open MPI (OMPI): Public MPI API, the only one which is exposed to applications.

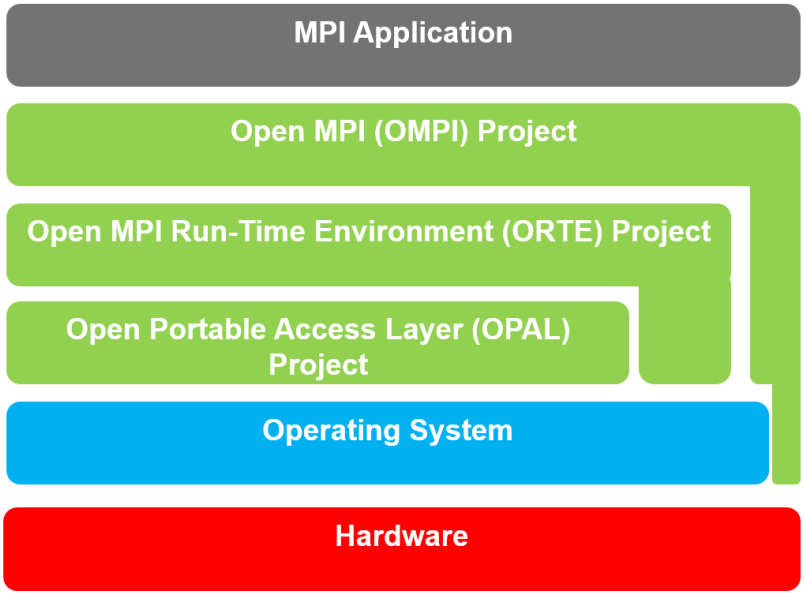

Figure A.5 OpenMPI High-Level View

The great variety of component frameworks in OpenMPI allows developers to use the specific platform as a deployment vehicle for commercial products. The deployment procedure is going to be investigated in the following subsection.

A.1.9. OpenMPI over RapidIO

As it has already been mentioned in Chapter 1, if the previously defined activities are completed within time a further investigation will also may be conducted on promoting openMPI to a complete framework over RapidIO for harvesting cooperative computational farms. Several MPI versions have already been employed on different interconnection fabrics. [16] presents, MPICH, an implementation of the MPI passing interface standard which can be deployed over Ethernet-connected workstations. As RapidIO is being used more and more as an interconnection fabric a research has been made for the deployment of an MPI library over RIO. [17] presents a release of the OpenFabrics Enterprise Distribution (OFED) to serial RapidIO, enabling, high-performance MPI applications to run on multicomputers.

Depending on the progress made with respect to the experimentation and the investigation that has been analysed in Chapter 5, time may be allocated for the deployment of openMPI techniques over RapidIO. More specifically, during the investigation that will be made in the multiple PDAK2Hs, if the procedure is completed well before the deadline and the experimental results meet the initially set goals, a research will be made about the communication between SRIO interconnected PDAK2Hs via OpenMPI. Figure A.6 illustrates an example of the topology that is going to be investigated.

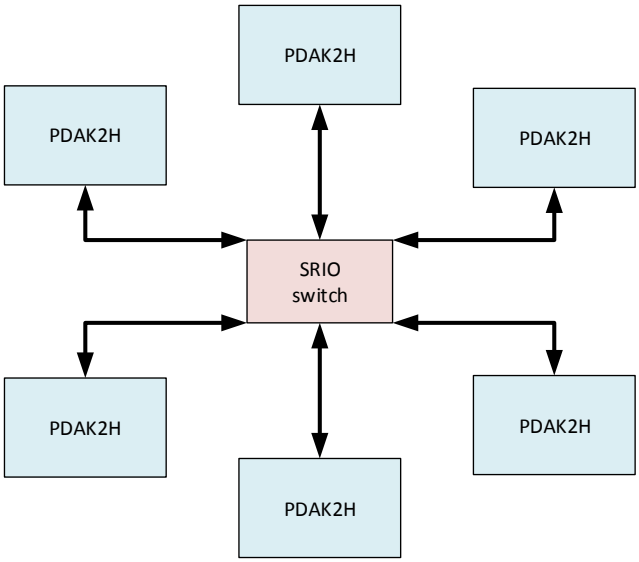

Figure A.6 A star topology involving PDAK2H nodes and a SRIO switch

[18] and [19] present the implementations that have already been developed by Prodrive Technologies and concern a hardware abstraction for RapidIO endpoint hardware and an abstraction layer on top of RapidIO, which implements low-level features like Direct Memory Access and sending/receiving of doorbells. In Figure A.7 an overview of the RapidIO software is presented. Hence, the goal for the future investigation will be a linkage between the OpenMPI library and the RapidIO Data Plane library.

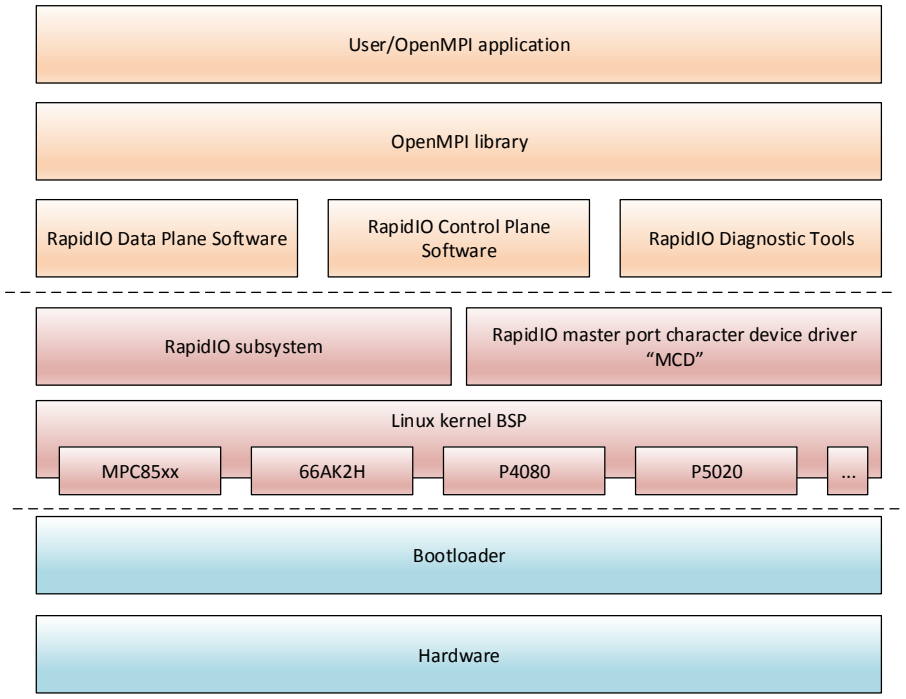

Figure A.7 Overview of the RapidIO software

Figure A.8 presents the architectural overview of OMPI project layer. The Byte Transfer Layer (BTL) is a framework which is responsible for data transfer between processes [22]. BTL API provides an abstraction over the available interconnect. For example, in Figure A.8 *sm* (shared memory) and *tcp* interconnections are presented. BTL API is presented in Table A.2. Table A.3 presents the supported by BTL devices.

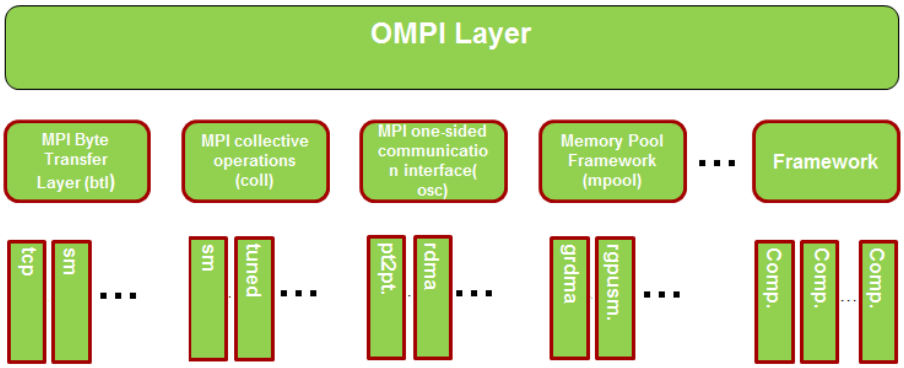

Figure A.8 OMPI Architectural Overview

Function	Description
btl_add_procs	Called to discover which processes are reachable by this module and create endpoint structures
btl_del_procs	Called to release resources held by the endpoint structures
btl_register	Called to register completion callback functions for a particular tag
btl_finalize	Called to release any resources held by the module
btl_alloc	Returns a BTL descriptor with free space but no actual data
btl_free	Releases a BTL descriptor and any memory used by it
btl_prepare_src	Returns a BTL descriptor that contains user data
btl_prepare_dst	Returns a BTL descriptor used for RDMA operations
btl_send	Initiates an asynchronous send of a particular BTL descriptor
btl_sendi	Initiates an immediate send of a particular BTL descriptor
btl_put	Initiates an asynchronous put (RDMA)
btl_get	Initiates an asynchronous get (RDMA)
btl_dump	Diagnostic information

Table A.2 Functions available in a BTL module

Interconnect	Description
openlib	Open Fabrics
self	send to self semantics
sm	shared memory
smcuda	shared memory with CUDA support
tcp	
ugni	Cray Gemini/Aries
vader	Shared memory using XPMEM/Cross-mapping

Table A.3 BTL supported devices

Hence, a future investigation regarding the deployment of OpenMPI over RapidIO would concern the development of a BTL which can support the RapidIO connectivity.

A.2. “Bones”

A brief analysis of the skeleton-based source-to-source compiler “Bones” follows. Its motivation as well as its details are described. Finally, the development of a backend for Bones tools for the mutli-SoC heterogeneous platforms is proposed.

A.2.1. Overview

The difficulty of programming heterogeneous platforms increases due to the differences between the programming approaches for multicore CPUs and DSPs/GPUs. The essentials of new parallel low level programming languages have to be considered and a detailed architectural knowledge is mandatory in order to proceed to an optimized application development. Due to the above difficulties, a research has been made regarding the auto-parallelization and auto-tuning. “Bones” is

a unique approach based on algorithmic skeletons which is able to automatically generate efficient and readable parallel code for parallel architectures (OpenMP, OpenCL or CUDA) [20].

The specific approach is based on “algorithmic species”, an algorithm classification technique targeted at parallel programming [21]. Species are classes of algorithms, capturing information such as data re-use and memory access patterns. The extraction of the aforementioned information forms the first step of the described approach and it is able to be done automatically through “ASET” [24] or “A-DARWIN” [21]. The second step consists of “Bones”, the combination of skeleton-based compilation with algorithmic species. Figure A.9 presents the two-step approach.

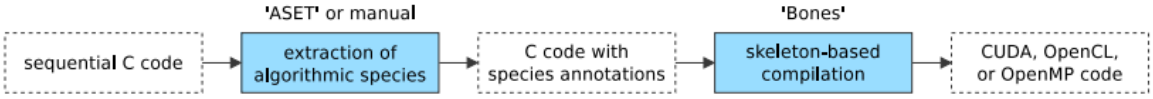

Figure A.9 Combining Skeletons with Algorithmic Species [20]

A.2.2. Algorithmic Species

The algorithm classification specified by algorithmic species is given based on access patterns of arrays in loop nests. The following code snippet illustrates an example of algorithmic species where the result of a matrix (`M`) – vector (`s`) multiplication is stored in a vector `r`. In order to produce a single element of `r` the following are needed:

- A complete row of matrix *M*
- The entire vector *s*

From the above, the final species for a matrix-vector multiplication is shown in the first line of the following snippet.

```
M[0:127, 0:63] | chunk(-,0:63) ^ s[0:63]|full → r[0:127]|element
for (i=0; i<128; i++) {
    r[i] = 0;
    for (j==0; j<64; j++) {
        r[i] += M[i][j] * s[j];
    }
}
```

A.2.3. Algorithmic Skeletons

Algorithmic skeletons can be described as templates for a specific group of computations on a specific platform. The following code snippet presents a simplified OpenMP skeleton (left) and how it can be configured (right) in order to serve the matrix-vector multiplication.

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
    int tid, i;
    int work, start, end;
    tid = omp_get_thread_num();
    work = <parallelism>/count;
    start = tid*work;
    end = (tid+1)*work;

    // Start the prallel work
    for (i=start; i<end; i++){
        <ids>
        <code>
    }

}
```

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
    int tid, i;
    int work, start, end;
    tid = omp_get_thread_num();
    work = 128/count;
    start = tid*work;
    end = (tid+1)*work;

    // Start the prallel work
    for (i=start; i<end; i++){
        int gid = i;
        r[gid] = 0;
        for (j=0; j<128; j++)
            r[gid] += M[gid][j]*s[j];
    }
}
```

Hence, as it can be seen, algorithmic species can be used for the determination of the most suitable skeleton. The algorithmic species, derived from the source C code, via a species-to-skeleton mapping are mapped to the available skeletons. Figure A. illustrates an example of “Bones” structure.

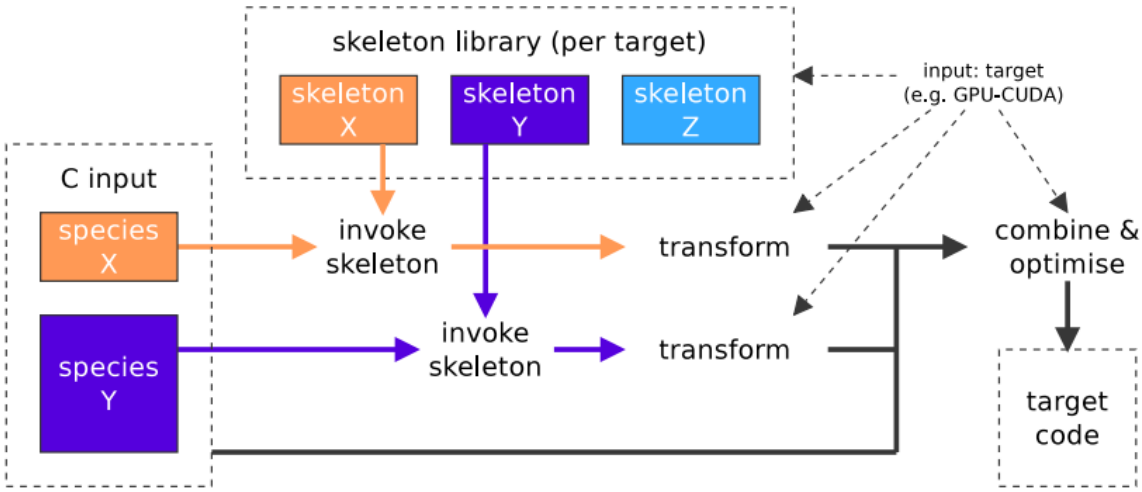

Figure A.10 Structure of "Bones" [20]

A.2.4. Backend for Bones tools for the Prodrive platform

Bones is able to produce output either for homogeneous platforms by generating OpenMP parallel code or output for platforms which combine CPUs and GPUs by generating OpenCL or CUDA. During the investigation that will be conducted, the feasibility of Bones to generate OpenCL or OpenMP code for a heterogeneous DSP architecture will be examined.

As it has already been mentioned in Chapter 2, during the experimental procedure, an investigation will be made regarding three different programming paradigms: ‘’plain-C”, OpenMP and OpenCL. The efficiency and the performance of the two parallel programming paradigms will be tested while the programming effort, which is needed for each one, will also be considered. By analysing the results of the aforementioned experiments, one of the two programming paradigms will be selected in order to create the corresponding skeletons. Hence, by using Bones with its new additional skeletons, a developer will be able to able to extract efficient and readable OpenMP or OpenCL code for a heterogeneous DSP platform, such as the one that has been described in Chapter 3.

Appendix B. K2H-PU Benchmarking tool

For the experimentation on the aforementioned configurations, the K2H-PU Benchmarking tool has been developed. The K2H-PU benchmarking tool has as a goal to benchmark and demonstrate the capabilities of processing units whose main component is the TI Keystone-II “Hawking” SoC. Two examples of such processing units are the PDAK2H and the ATCA-TK2-6PU blade.

The user is able to give as input a compute-intensive algorithm and the input-data that are going to be processed. The programming method (“plain-C”, openMP, openCL described in Chapter 4) and the K2H-PU configuration (PDAK2H or ATCA-TK2-6PU with the specified number of ARMs, DSPs, and PUs) are the parameters that can be configured by the user. After the execution of the algorithm, the output-data and the aggregated results are given by the tool as output.

Finally, the user is able to use the tool both remotely from a host Windows/Linux machine and locally from a processing unit. Figure B.Figure 6.1 presents an overview of the graphical user interface (GUI) of the tool.

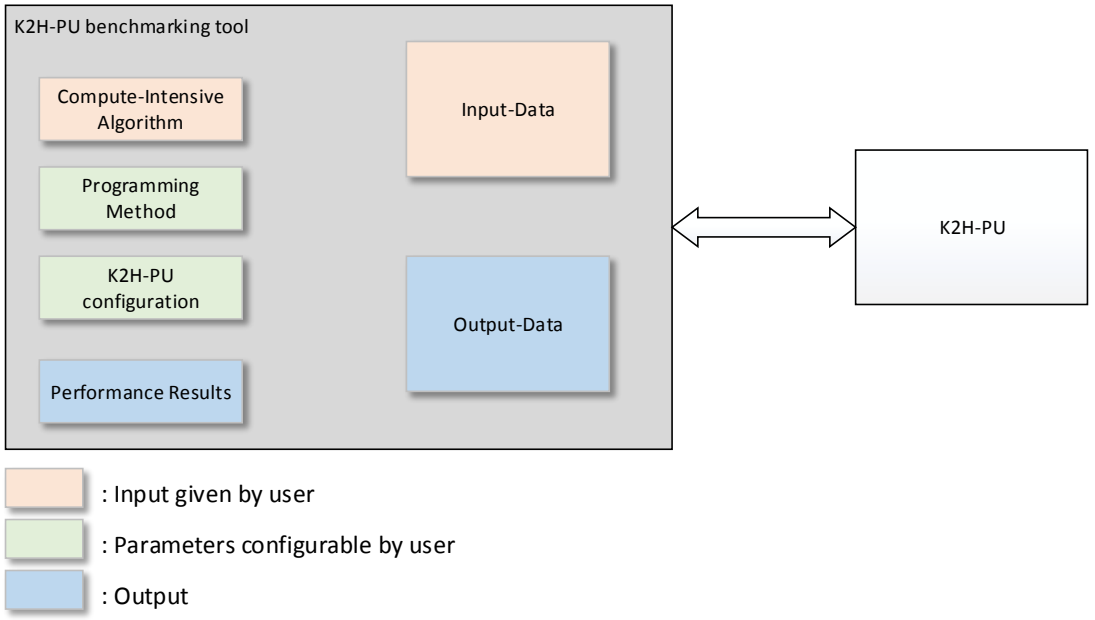

Figure B.1 K2H-PU GUI

B.1. Functional design

When the user specifies from a host machine through the GUI the processing units and the rest of the available parameters, a C application, called Master Task Manager, is configured automatically by the GUI. When the configuration is completed, the user is able to run the selected application.

The Master Task Manager configures a number of C applications, called Slave Task Managers, which are stored in the selected processing units and they are responsible for the execution of the application.

The Slave Task Manager can either be called remotely by the Task Manager via *ssh command* or directly by the user who is logged in the specific processing unit and would like to run the application only in this processing unit.

Hence, each processing unit has to store:

1. The Slave Task Manager (C application)
2. The application (ELF file)
3. The input data in the case of a standalone execution

Figure B.2 represents the above.

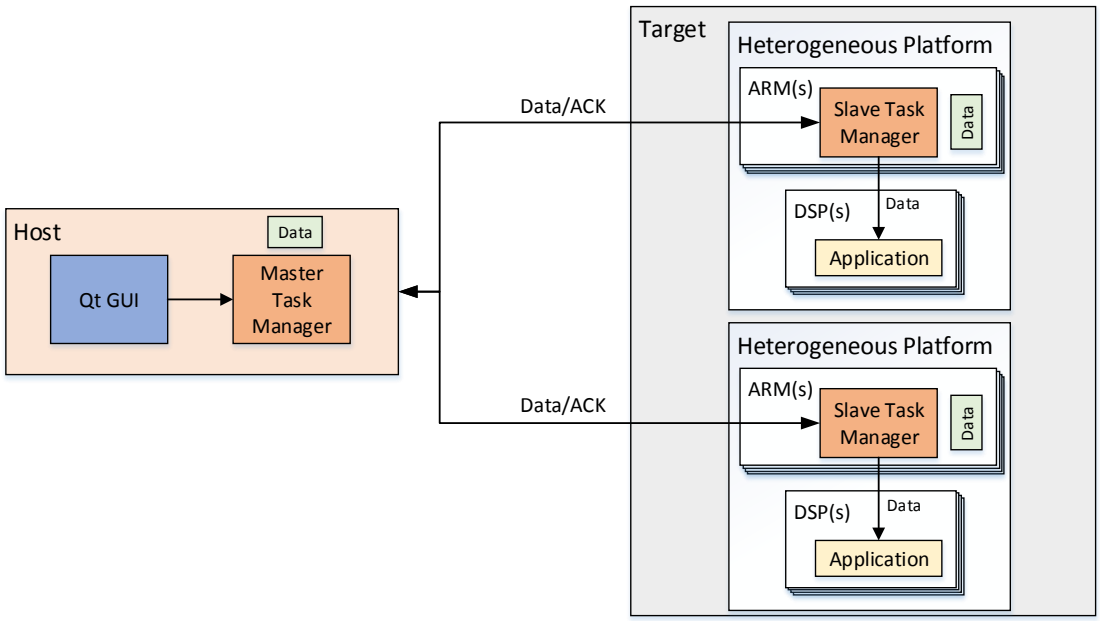

Figure B.2 Functional design of the K2H-PU benchmarking tool

B.1.1. Master Task Manager

B.1.1.1. Features

Master Task Manager is a C application which runs on both Linux and Windows OS. In the current version a wired connection between the Master Task Manager (x86 host machine) and the processing units is required. In addition to the GUI, a command line user interface is provided.

B.1.1.2. Responsibilities

The Task Manager is responsible for:

1. Dividing the input data
2. Distributing the input data parts to the available processing units
3. Triggering the execution of the Slave Task Manager
4. Retrieving the output data from the available processing units

B.1.2. Slave Task Manager

B.1.2.1. Features

Slave Task Manager is a C application which runs on Linux OS. A wired connection via the hyperlink interface between the Slave Task Manager (HWK ARMs) and the off-chip DSPs is required. A command line user interface is provided.

B.1.2.2. Responsibilities

The Slave Task Manager is responsible for:

1. Dividing the input data
2. Distributing the input data parts to the available ARMs/DSPs
3. Executing the application
4. Retrieving the output data from the available ARMs/DSPs

B.1.3. Communication

B.1.3.1. Host-Target Communication

The inter-process communication and the data transfer between the Task Manager and the Slave Task Manager uses the client server model and it is achieved via datagram (UDP) sockets in the

Internet Domain. Due to the small distance between the host and the PUs a packet loss was not noticed during the performed experiments. *Scp* could be an alternative method for the data transfer. However, due to the handling mechanisms that are provided in *scp* a much lower performance was achieved. The Master Task Manager triggers the execution of the Slave Task Manager via *ssh command*.

The flow-diagram of the tool is presented in Figure B.3.

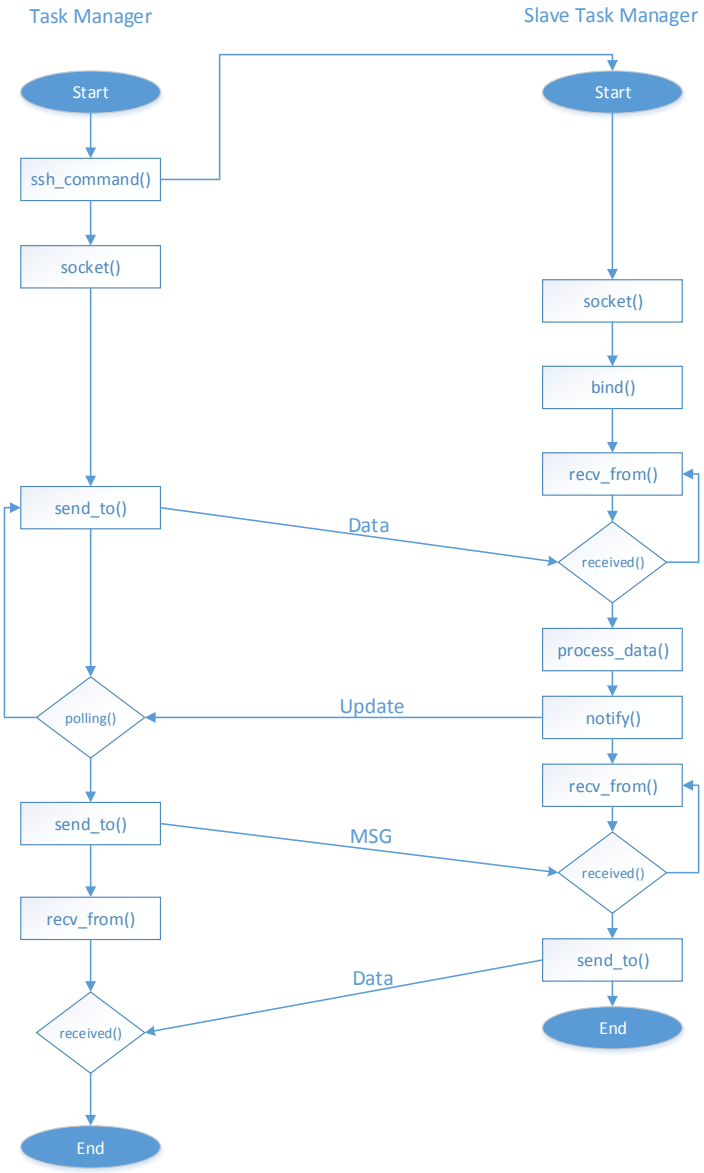

Figure B.3 K2H-PU benchmarking tool flow-diagram

As it is mentioned above the Slave Task Manager is responsible for the execution of the application. Hence after a successful ssh connection between the Master and the Slave Task Manager, it is feasible to end the ssh connection while the launched application continues its execution. The above is achieved by daemon-izing the Slave Task Manager.

The Task Manager is able to call the Slave Task Managers of each processing unit either simultaneously or sequentially depending on the type of the application. In the current version the Slave Task Managers are called sequentially.

B.1.3.2. Intra-target communication

The Slave Task Manager is executed in the ARM side of the processing unit. As it is mentioned in Chapter 4, the communication with the available DSPs and the execution of the application in the DSP side can be achieved by three different programming methods:

1. “plain-C”
2. OpenMP
3. OpenCL

In the “plain-C” method the application is parallelized manually and the communication with the available DSPs can be achieved via the MPM transport library [35]. By using the OpenMP or the OpenCL programming method it is not feasible to program the extra sixteen DSPs offered by the two Shannons. Only the eight DSPs offered by the Hawking SoC can be programmed.

B.2. Operating principle

B.2.1. Master Task Manager

By default no processing units are specified to be used and benchmarked. To include a processing unit, command line arguments can be supplied to the Master Task Manager.

Master Task Manager starts by diving equally the input data to the selected processing units. When a packet of data is ready, it is distributed to one of the processing units. The data transfer is performed via UDP sockets.

After the successful data distribution, the Master Task Manager waits for the termination of the data process by the complete set of the processing units. It is notified about the status of the processing units by performing a polling operation. It samples continuously log files which are updated by the Slave Task Managers. The sampling procedure is accomplished by using the Secure Copy (SCP) protocol.

The final step of the Master Task manager is the retrieval of the output data from the processing units. The retrieval is achieved again via UDP sockets.

B.2.1.1. Master Task Manager parameters

Table B.1 presents the command line parameters of the Master Task Manager.

Parameter	Allowed Values	Required	Default	Remark
`-p`	Reachable IP(s)	Yes	-	Defines the IP(s) of the reachable K2H-PU(s)
`-c`	HWK, SHN0, SHN1	No	HWK	Defines the SoC of the specified K2H-PU(s)
`-d`	1 – 8	No	8	Follows the `-c` parameter and defines the number of the available DSPs of the specified SoC
`-a`	1 – 4	No	4	Follows the `-c` parameter and can be set only when the HWK SoC has been specified. It defines the number of the available ARMs
`-m`	OpenCL, OpenMP, plainC	No	plainC	Defines the programming method that is used. If the OpenCL or the OpenMP option is selected the `-c`, `-d` and `-a` options are ignored. Only the 8 DSPs of the HWK SoC are used
`-h`	-	No	-	Displays the help page
`-x, -y`	>0	Yes	-	Both parameters depend on the input data. In the current application they specify the dimensions of the input image

Table B.1 Master Task Manager parameters

In the following example a 1024x768 image is divided in two processing units (15.2.5.25, 15.2.5.50). In each processing unit a Hawking with 8 DSPs and 4 ARMs and two Shannons with 8 DSPs each are used. “plain-C” is the selected programming method.

Command line example: `./master_task_manager -x 1024 -y 768 -p 15.2.5.25 15.2.5.50 -c HWK -d 8 -a 4 -c SHN0 -d 8 -c SHN1 -d 8 -m plainC`

B.2.1.2. Master Task Manager output

The following results are given by the Master Task Manager:

- Transfer Time from the host x86 machine to the available processing units
- Computation Time
- Transfer Time from the available processing units to the host x86 machine
- Total execution Time (the sum of the previous three measurements)
- The theoretical optimal configuration for the specific application and input data, based on the analytical performance models (Chapter 6.5), and the relevant times
- The transfer and computation times if all the available processing units were used

Hence the user is able to compare its configuration with the theoretical optimal one and the one which uses all the available resources.

B.2.2. Slave Task Manager

By default no SoCs are specified to be used and benchmarked. To include a SoC, command line arguments can be supplied to the Slave Task Manager.

Two versions of the Slave Task Manager are provided. The first is a standalone application which is used to benchmark and demonstrate a single K2H-PU. The user has to login in a specific K2H-PU in order to execute this version. The second version is used to benchmark and demonstrate several K2H-PUs and it is combined with the Master Task Manager.

The main difference between the two versions is that the first one is able to read the input data by itself while the second one retrieves the input data from the Master Task Manager via UDP sockets (it also sends the output data to the Master Task Manager again via UDP sockets). The second version has also to notify the Master Task Manager about the successful process of the input data. This is done by updating a log file which is stored in the PU.

Both versions of the Slave Task Manager divide equally the input data to the selected SoCs. When a packet of data is ready, it is distributed to one of the SoCs. The target region of memory which is used for the store of the data in order to be processed by the specified SoCs depends on the size of the input data. In the case where the input data are less than 6Mbytes the shared on-chip memory is used. It is accessed either using the `mmap()` function for the HWK SoC or using the `mpm_transport_mmap()` for the two Shannons SoCs. If the input data are more than 6Mbytes the DDR off-chip memory has to be used. If the Shannon SoCs has been selected the input data have to be divided and distributed in chunks of maximum 4Mbytes because of the limitation of `mpm_transport_mmap()` to map regions greater than 4Mbytes. Instead of `mpm_tranport_mmap()`, `mpm_transport_write()` or `mpm_transport_put_initiate()` can be used for faster transfers.

DSP loader library [6] is used to:

1. reset the specified DSP cores
2. load the application (ELF file) into the physical DSP memory and
3. de-assert reset of the specified DSP cores in order to start the execution of the DSP application

Slave Task Manager is notified about the successful process of the data by the DSPs via the polling procedure. It samples continuously a number of memory addresses which are updated by the relevant DSP core.

By using the same functions as above, Slave Task Manager access the correct memory regions and retrieves the output data from the DSP cores.

B.2.2.1. Slave Task Manager parameters

Table B.2 presents the command line parameters of the Slave Task Manager.

Parameter	Allowed Values	Required	Default	Remark
`-c`	HWK, SHN0, SHN1	No	HWK	Defines the SoC of the specified K2H-PU(s)

-d	1 – 8	No	8	Follows the -c parameter and defines the number of the available DSPs of the specified SoC
-a	1 – 4	No	4	Follows the -c parameter and can be set only when the HWK SoC has been specified. It defines the number of the available ARMs
-m	OpenCL, OpenMP, plainC	No	plainC	Defines the programming method that is used. If the OpenCL or the OpenMP option is selected the -c, -d and -a options are ignored. Only the 8 DSPs of the HWK SoC are used
-h	-	No	-	Displays the help page
-x, -y	>0	Yes	-	Both parameters depend on the input data. In the current application they specify the dimensions of the input image. It is used only in the standalone version.

Table B.2 Slave Task Manager parameters

In the following example a Hawking with 8 DSPs and 4 ARMs and two Shannons with 8 DSPs are used for the process of a 1024x768 image. “plain-C” is the selected programming method.

Command line example: `./slave_task_manager -x 1024 -y 768 -c HWK -d 8 -a 4 -c SHN0 -d 8 -c SHN1 -d 8 -m plainC`

B.2.2.2. Slave Task Manager output

The following results are given by the Slave Task Manager:

- Transfer Time from the host (HWK ARMs) to the available DSPs
- Computation Time
- Transfer Time from the available DSPs to the host (HWK ARMs)
- Total execution Time (the sum of the previous three measurements)
- The theoretical optimal configuration for the specific application and input data, based on the analytical performance models (Chapter 6.5), and the relevant times
- The transfer and computation times if all the available DSPs and ARMs were used

Hence the user is able to compare its configuration with the theoretical optimal one and the one which uses all the available resources.

References

[1] OpenMP Architecture Review Board “OpenMP Application Program Interface”, 2013

[2] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2012

[3] Khronos OpenCL Working Group “The OpenCL Specification”, 2014

[4] Prodrive Technologies “SPD of PDAK2H (Prodrive DSP AMC Keystone-II Hawking)”, 2014

[5] Prodrive Technologies “SPD of APBC66: AMC Piggy Back C66x”, 2014

[6] Prodrive Technologies “SPD of WING-IP (Wafer Inspection Next Generation – Image Processor)”, 2014

[7] Lindenbaum, Michael, M. Fischer, and A. Bruckstein. "On Gabor's contribution to image enhancement." *Pattern Recognition* 27.1,1994

[8] Perona, Pietro, and Jitendra Malik. "Scale-space and edge detection using anisotropic diffusion." *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 12.7, 1990

[9] Buades, Antoni, Bartomeu Coll, and Jean-Michel Morel. "A review of image denoising algorithms, with a new one." *Multiscale Modeling & Simulation* 4.2, 2005

[10] Lebrun, Marc. "An analysis and implementation of the BM3D image denoising method." *Image Processing On Line* 2012, 2012

[11] Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52.4, 2009

[12] Fuller, Sam. *RapidIO: The embedded system interconnect*. John Wiley & Sons, 2005

[13] Greg Shippen “System Interconnect Fabrics: Ethernet versus RapidIO Technology”, 2007

[14] Porter, Matt. "RapidIO for Linux." *Linux Symposium*. 2005

[15] Gabriel, Edgar, et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation." *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2004

[16] Gropp, William, et al. "A high-performance, portable implementation of the MPI message passing interface standard." *Parallel computing* 22.6, 1996

[17] Cain, Kenneth. "Performance Migration to Open Solutions: OFED for Embedded Fabrics.", 2010

[18] Prodrive Technologies “SPD of MPort Driver”, 2014

[19] Prodrive Technologies “SPD of RapidIO Data Plane Library”, 2014

[20] Nugteren, Cedric, Pieter Custers, and Henk Corporaal. "Automatic skeleton-based compilation through integration with an algorithm classification." *Advanced Parallel Processing Technologies*. Springer Berlin Heidelberg, 2013

[21] Nugteren, Cedric, Rosilde Corvino, and Henk Corporaal. "Algorithmic species revisited: A program code classification based on array references." *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*. IEEE, 2013

[22] Hayes, Timothy. "An Efficient Open MPI Transport System for Virtual Worker Nodes", 2009

[23] Prodrive Technologies “Software User Manual of PDAK2H (Prodrive DSP AMC Keystone-II Hawking)”, 2013

[24] Custers, P. J. J. M. *Algorithmic species: Classifying program code for parallel computing*. Diss. Master’s thesis, Eindhoven University of Technology, 2012

[25] Berkeley Design Technology, Inc. “Evaluating DSP Processor Performance”, 2002

[26] Zivojnovic, Vojin, et al. "DSPstone: A DSP-oriented benchmarking methodology." *Proceedings of the International Conference on Signal Processing Applications and Technology*. 1994

[27] Mohamed, Ahmed M., Lester Lipsky, and Reda A. Ammar. "Performance Modeling of a Cluster of Workstations." *Communications in Computing*. 2003

[28] Khanyile, Nontokozo P., Jules-Raymond Tapamo, and Erick Dube. "An analytic model for predicting the performance of distributed applications on multicore clusters.", 2012

[29] Khanyile, Nontokozo P., Jules-Raymond Tapamo, and Erick Dube. "An analytic model for predicting the performance of distributed applications on multicore clusters.", 2012

[30] Wang, Li, et al. "Efficient task assignment on heterogeneous multicore systems considering communication overhead." *Algorithms and Architectures for Parallel Processing*. Springer Berlin Heidelberg, 2012

[31] Kim, Hanjun, et al. "Automatic speculative doall for clusters." *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012

[32] Amini, Mehdi, et al. "Par4all: From convex array regions to heterogeneous computing." *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012

[33] Texas Instruments “66AK2H14/12/06 Multicore DSP+ARM Keystone II System-on-Chip”, 2013

[34] OpenMP Accelaratior Model 0.3.3, http://processors.wiki.ti.com/index.php/OpenMP_Accelerator_Model_0.3.3#.23pragma_omp_declare_target

[35] Prodrive Technologies “SPD of Linux Hyperlink driver”, 2014

[36] Hoefler, Torsten. "Bridging performance analysis tools and analytic performance modeling for HPC." Euro-Par 2010 Parallel Processing Workshops. Springer Berlin Heidelberg, 2011.