Low-power System-on-Chip Processors for Energy Efficient High Performance Computing: The Texas Instruments Keystone II

Gaurav Mitra

A thesis submitted for the degree of
Doctor of Philosophy (Computer Science)
The Australian National University

October 2017

© Gaurav Mitra 2017

Except where otherwise indicated, this thesis is my own original work.

Gaurav Mitra
24 October 2017

To Maa, Baba and Arya

Acknowledgments

First, I would like to thank my supervisor, Prof. Alistair Rendell for his invaluable guidance and encouragement throughout my time at the ANU. This thesis would not have been possible without Alistair's continued support and unwavering patience.

I would like to thank Dr. Eric McCreath, Dr. Josh Milthorpe, Dr. Ashvin Parameswaran and Dr. Joseph Antony for reviewing parts of my thesis and providing useful feedback. My colleagues, Anish Varghese and Arindam Sharma also reviewed portions of my thesis, for which they have my gratitude.

The Australian National University and the Australian Government have supported me financially with an undergraduate scholarship, an Australian Postgraduate Award and numerous teaching opportunities throughout the duration of my studies in Australia. I am deeply grateful. I would also like to thank CSIRO for the financial support I received from the ICT Centre Postgraduate Research Scholarship during the first six months of my PhD. In addition, I would like to acknowledge financial support from the Australian Research Council Discovery Project (DP0987773).

Major elements of this work were made possible with support from Texas Instruments Inc. In particular, I would like to thank my managers at TI, Ajay Jayaraj and Dr. Eric Stotzer for their guidance throughout my internship at TI. My work with nCore HPC enabled significant parts of this work. I would like to thank Ian Lintault for providing me with the opportunity to work on the nCore BrownDwarf system.

My time at the ANU has been very enjoyable and in particular with the Computer Systems Group and the Research School of Computer Science. I have had spirited discussions with many colleagues at the university which have led to new and interesting ideas. I would like to thank all my friends and colleagues at the ANU for enriching my time here.

For final months of my PhD, I was employed full-time at the National Computational Infrastructure where I found immense encouragement. In particular, I would like to thank Dr. Roger Edberg and Dr. Muhammad Atif.

Finally, I would like to thank my mother, Dr. Rupa Mitra; my father, Dr. Debasis Mitra; and my wife Arya Bhattacharjee for their endless love, inspiration, and optimism. This journey would not have been possible without them.

Abstract

The High Performance Computing (HPC) community recognizes energy consumption as a major problem. Extensive research is underway to identify means to increase energy efficiency of HPC systems including consideration of alternative building blocks for future systems. This thesis considers one such system, the Texas Instruments Keystone II, a heterogeneous Low-Power System-on-Chip (LPSoC) processor that combines a quad core ARM CPU with an octa-core Digital Signal Processor (DSP). It was first released in 2012.

Four issues are considered: i) maximizing the Keystone II ARM CPU performance; ii) implementation and extension of the OpenMP programming model for the Keystone II; iii) simultaneous use of ARM and DSP cores across multiple Keystone SoCs; and iv) an energy model for applications running on LPSoCs like the Keystone II and heterogeneous systems in general.

Maximizing the performance of the ARM CPU on the Keystone II system is fundamental to adoption of this system by the HPC community and, of the ARM architecture more broadly. Key to achieving good performance is exploitation of the ARM vector instructions. This thesis presents the first detailed comparison of the use of ARM compiler intrinsic functions with automatic compiler vectorization across four generations of ARM processors. Comparisons are also made with x86 based platforms and the use of equivalent Intel vector instructions.

Implementation of the OpenMP programming model on the Keystone II system presents both challenges and opportunities. Challenges in that the OpenMP model was originally developed for a homogeneous programming environment with a common instruction set architecture, and in 2012 work had only just begun to consider how OpenMP might work with accelerators. Opportunities in that shared memory is accessible to all processing elements on the LPSoC, offering performance advantages over what typically exists with attached accelerators. This thesis presents an analysis of a prototype version of OpenMP implemented as a bare-metal runtime on the DSP of a Keystone I system. An implementation for the Keystone II that maps OpenMP 4.0 accelerator directives to OpenCL runtime library operations is presented and evaluated. Exploitation of some of the underlying hardware features of the Keystone II is also discussed.

Simultaneous use of the ARM and DSP cores across multiple Keystone II boards is fundamental to the creation of commercially viable HPC offerings based on Keystone technology. The nCore BrownDwarf and HPE Moonshot

systems represent two such systems. This thesis presents a proof-of-concept implementation of matrix multiplication (GEMM) for the BrownDwarf system. The BrownDwarf utilizes both Keystone II and Keystone I SoCs through a point-to-point interconnect called Hyperlink. Details of how a novel message passing communication framework across Hyperlink was implemented to support this complex environment are provided.

An energy model that can be used to predict energy usage as a function of what fraction of a particular computation is performed on each of the available compute devices offers the opportunity for making runtime decisions on how best to minimize energy usage. This thesis presents a basic energy usage model that considers rates of executions on each device and their active and idle power usages. Using this model, it is shown that only under certain conditions does there exist an energy-optimal work partition that uses multiple compute devices. To validate the model a high resolution energy measurement environment is developed and used to gather energy measurements for a matrix multiplication benchmark running on a variety of systems. Results presented support the model.

Drawing on the four issues noted above and other developments that have occurred since the Keystone II system was first announced, the thesis concludes by making comments regarding the future of LPSoCs as building blocks for HPC systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Performance Computing (HPC) seeks to solve complex real world problems, such as material design, climate science and financial modeling, in a timely fashion. Leading supercomputing systems are massively parallel, containing hundreds of thousands of nodes that are themselves composed of many individual processing elements. They are also becoming increasingly heterogeneous, in that each node often combines several conventional CPUs with a number of attached accelerator devices, such as Graphics Processing Units (GPU). Additionally, the embedded HPC space is rapidly expanding with artificial intelligence (AI) and computer vision leading the way. For example, autonomous and semi-autonomous vehicles available today require significant amounts of processing capability on-board in order to perform real-time scene recognition and object detection.

Both these communities, HPC and embedded, are facing the common challenge of energy consumption. At the high end are warehouse scale modern supercomputing systems. The ongoing cost and environmental impact of their energy usage and cooling requirements are rapidly becoming a major problem [Attig et al., 2011]. At the other end embedded solutions, particularly those operating in a mobile environment, are also constrained on a limited energy budget while being faced with ever increasing computing workloads. It is therefore not surprising that the HPC community is investing heavily in research to increase the energy-efficiency of current HPC systems along with the exploration of alternative energy-efficient building blocks for future *exascale* [Bergman et al., 2008; Shalf et al., 2011] HPC systems.

The primary metric used to measure energy-efficiency in the HPC domain is double-precision floating-point operations per second per watt i.e. FLOPS/watt. In 2012, Dongarra and Luszczek [2012] performed experiments on an Apple iPad 2 housing a dual-core ARM Cortex-A9 processor. Their observations led to a three-tier categorization of system performance based on energy-efficiency: i) 1 GFLOPS/watt: desktop and server processors, ii) 2 GFLOPS/watt: attached

GPU accelerators, and iii) 4 GFLOPS/watt: ARM Cortex-A processors (where 1 GFLOPS equals 10^9 floating point operations per second). At the time the vast majority of HPC systems were based on x86 technology possibly with attached GPU accelerators, and ARM processors had little penetration in this market; in spite of the fact that ARM Holdings reported sales of over 8 billion devices in that year [ARM Ltd., 2013].

The mainstream market success of the ARM architecture indicated a paradigm shift in computing with ARM systems replacing x86 [Rajovic et al., 2013a]. Interestingly, the x86 architecture, however, had previously been the beneficiary of a similar paradigm shift. Between year 1990 and 2000, HPC hardware experienced this shift. Complex instruction set (CISC) commodity micro-processors replaced special purpose vector, reduced instruction set (RISC), and SIMD processors primarily because of economic factors such as the commodity cores being 30× cheaper than its vector counterparts. As the commodity processors had significantly lower per-processor performance than vector ones, this transition was made possible by new programming models such as the message passing interface (MPI) that allowed multiple commodity processors to work together over a network, initiating the so called *Beowulf cluster* revolution [Becker et al., 1995].

Today, the majority of ARM processors are found in mobile devices such as smart-phones and tablets. These devices are invariably composed of CPU cores alongside accelerator cores such as Graphics Processing Units (GPU) in so called heterogeneous Low Power System-on-Chip (LPSoC) processors.

Intel has embraced this paradigm shift and has recently been manufacturing LPSoCs aimed for mobile devices consuming under 4.5 watts [Intel Corporation, 2014]. This economic driver, the need to constrain energy consumption in very high end HPC systems, and increasing environmental concerns around the energy usage in large scale data centers, will inevitably lead to the widespread adoption of cheap low-power processors in all future HPC systems. The question is simply when [Rajovic et al., 2013a].

Interestingly, processors being used in current supercomputers that have made significant improvements in energy-efficiency, have also witnessed a shift towards use of multiple low power RISC cores similar to the ARM cores evaluated by Dongarra and Luszczek [2012]. The fastest supercomputer in the world as of June 2017, the *Sunway TaihuLight* system [Top500], uses custom designed homogeneous 260-core SW26010 [Fu et al., 2016] CPUs composed of low-power RISC cores. These RISC cores bear close resemblance to typical accelerator architectures such as that of a GPU Streaming Multiprocessor (SM) or a many-core Intel Xeon-Phi processor, housing hundreds of very low-power compute cores. It performs at 93 PFLOPS while consuming 15.37 Mwatts of power and has an

energy-efficiency of 6.05 GFLOPS/watt.

The most energy-efficient supercomputer in the world, the *TSUBAME 3.0* [Green500] uses attached GPU accelerators and performs at 14.1 GFLOPS/watt. The TSUBAME 3.0 comprises Intel Xeon CPUs and NVIDIA Tesla P100 GPU cards. Such PCiE attached GPU accelerators, however, suffer from a serious problem of their on-board memory being physically separate to the host CPU memory which is often responsible for mandatory memory transfer delays during computation. In spite of this disadvantage, attached GPU systems have high energy-efficiency.

A glaring similarity between both the TaihuLight and the TSUBAME 3.0 state-of-the-art systems is the use of many low-power cores. In fact, a majority of other systems in the current Top500 and Green500 lists use accelerator cards composed of thousands of low-power cores. Nine out of the top ten systems in the June 2017 Green500 list are composed of NVIDIA P100 GPUs. The other system in the top 10 on this list, the *Gyoukou*, is composed of a many-core PEZY-SC2 CPU similar to the Sunway TaihuLight many-core CPU.

Looking towards future supercomputers, the next Fujitsu *Post-K* [Fujitsu, 2016] supercomputer, due to be operational in 2020, is set to incorporate 64-bit ARM v8 based processors specifically designed for HPC. Another novel future HPC system, *The Machine* [Courtland, 2016], is set to be manufactured by HPE. It is designed to tackle massive memory bound problems by providing a global shared address space memory in the order of Petabytes and aims to herald the so called *memory-driven* computing era. The first iteration of The Machine is also composed of ARM v8 SoC processors. Other notable initiatives such as the European MontBlanc project [Grasso et al., 2014; Rajovic et al., 2013c] are pursuing similar directions.

With the evolution and tremendous market success of mobile computing, low-power processors have witnessed a meteoric rise. Heterogeneous LPSoC processors composed of CPU cores along with on-chip accelerator cores in particular have shown ample promise of energy-efficiency. They are fundamental components of all contemporary smartphones and portable devices where battery power consumption is the foremost concern. LPSoCs also do not suffer from the problem of physically separate memory between host and accelerator which PCiE attached accelerators face today. Being heterogeneous by design, LPSoCs provide specialized processing elements equipped to tackle different parts of a single problem. In essence, an SoC combines all the elements of a traditional computer on a single chip. That is, it may combine a traditional CPU, an accelerator such as GPU, main memory and memory subsystem controllers, a variety of network interfaces and other application specific co-processors. Compared to the use of separate devices (e.g. CPUs with attached accelerators), SoCs offer

significant power, reliability, and speed advantages.

In 2011, the Texas Instruments Keystone I LPSoC composed of an octa-core C66x digital signal processor (DSP) was demonstrated to provide 7.4 GFLOP-S/watt [Igual et al., 2012] of energy-efficiency. In 2012, the Keystone II SoC was introduced. It integrated the C66x octa-core DSP with a quad core general purpose ARM Cortex-A15 processor and a variety of other devices.

Targeted towards the embedded signal processing market, one of the Keystone II SoC's most important design features was its ability to perform double-precision floating-point operations. As such it promised higher energy efficiency compared to supercomputers in the Green500 list [Green500] and therefore raised significant interest in the HPC community.

Whether LPSoC processors are suitable for HPC applications dominated by floating-point calculations and whether they can actually be programmed in an energy-efficient and portable manner are open questions. The overall objective of this work was to evaluate the suitability of the Keystone II architecture for HPC applications. This included the design and implementation of its HPC programming environment.

The work presented in this thesis began in 2012 as a collaboration with TI. A portion of this work was completed alongside compiler engineers at TI offices in Houston, Texas. Key contributions from this work also enabled nCore HPC, a Silicon Valley start-up, to bring to market the first HPC system using both the Keystone I and Keystone II SoCs, the *BrownDwarf* in 2013.

Several issues exist that have prevented LPSoCs such as the Keystone II from being readily adopted by the HPC ecosystem. Contributions of this thesis addressing these specific issues include:

ARM CPU Performance. This thesis demonstrates the use of compiler intrinsic functions to improve ARM NEON code vectorization and application performance across four generations of ARM processors. Comparisons are drawn with Intel SSE2 operations.

Suitability of OpenMP for C66x DSP Accelerator. This thesis evaluates the OpenMP HPC programming model on C66x multi-core DSP and demonstrates the first successful mapping of OpenMP 4.0 constructs to OpenCL runtime library operations through an implementation of the OpenMP 4.0 runtime on Keystone II SoC.

Commercial proof-of-concept. This thesis demonstrates the simultaneous use of both Keystone I and Keystone II SoCs for Level-3 BLAS matrix multiplication through the implementation of a message passing communica-

tion framework for the first commercially available system using Keystone SoCs, the nCore BrownDwarf system.

Measuring energy consumption. This thesis demonstrates building a novel high resolution energy measurement environment for LPSoC systems that enables measurement of energy similar to measurement of process time.

Modeling energy consumption. This thesis defines an energy usage model for heterogeneous systems such as LPSoCs. This model factors in the cost of using multiple processing elements simultaneously for computation through work partitioning.

Predicting an energy-optimal work partition. Using the energy model this thesis shows that an energy-optimal work partition that uses multiple compute devices, does not always exist on a heterogeneous system. It further defines an equation to apriori predict its existence. This is validated using multiple LPSoC systems alongside current Intel and NVIDIA GPU based HPC systems.

Demonstrating application performance and energy-efficiency. This thesis demonstrates use of the Keystone II SoC for Level-3 BLAS matrix multiplication, a critical HPC building block. Measured energy efficiencies with comparisons to other contemporary LPSoC systems as well as state-of-the-art Intel based HPC systems are also reported.

1.1 Chapter Summary

Chapter 2 provides a detailed literature review covering features of processing elements present in LPSoC systems. Programming models including OpenMP and OpenCL are described alongside technical aspects of their runtime environments. Related work spanning all facets of this thesis is also provided.

Chapter 3 presents the first detailed analysis and evaluation of ARM NEON SIMD instructions contrasted against Intel SSE SIMD instructions and presents a case for using compiler intrinsic functions to maximize performance, while maintaining code portability.

The majority of performance on a contemporary CPU for any application is dependent on Single Instruction Multiple Data (SIMD) operations. Techniques to maximize the effectiveness of SIMD operations on ARM CPU

cores are crucial to extracting maximum performance from the TI Keystone II and LPSoCs in general. Chapter 3 also presents insights into how data size affects ARM CPU performance while comparing across four generations of ARM CPUs.

Chapter 4 presents the first evaluation of an alpha version of TI's *bare-metal* OpenMP runtime on the Keystone II SoC's C66x octa-core DSP with comparisons drawn against other OpenMP runtimes on contemporary Intel HPC systems as well ARM systems. This chapter also presents the first implementation of the OpenMP 4.0 runtime environment on the TI Keystone II, in fact the first on any LPSoC.

The C66x octa-core DSP used in the TI Keystone II SoC was primarily designed and implemented for signal processing workloads in wireless basestations along with other embedded computing scenarios. Traditionally, embedded computing domains used hand-optimized codes, often programmed in assembly language, intended to perform a single task with the best possible efficiency throughout the lifetime of the system. As a consequence programming models used in HPC such as OpenMP, OpenCL and MPI have generally not been supported by LPSoC manufacturers and programmers. To bring the C66x DSP within reach of HPC programmers, TI implemented a bare-metal OpenMP runtime for the octa-core C66x which is evaluated in this work.

In 2013, the OpenMP 4.0 specification was released which proposed accelerator specific *target* directives designed to offload computation from *host* CPU cores to *target* accelerator cores. The implementation of OpenMP 4.0 for Keystone II presented in this work takes advantage of shared physical memory between ARM and DSP cores. A new OpenMP construct, the *local* keyword, is also proposed to make use of fast local memory available to DSP cores.

Chapter 5 presents the first and only available implementation of a HPC programming environment for the nCore BrownDwarf system. Using this environment, a proof-of-concept implementation and evaluation of Level-3 BLAS matrix multiplication (GEMM) is also presented across a cluster of four BrownDwarf nodes.

Each node of a BrownDwarf system is composed of a single Keystone II SoC along with two Keystone I SoCs connected via the TI Hyperlink interconnect. A new communication framework between the Keystone II ARM cores and Keystone I DSP cores is implemented using a message mass-

ing protocol. Using this framework, Level-3 BLAS matrix multiplication is implemented to utilize all processing elements on BrownDwarf and its performance is measured and evaluated across four nodes.

Chapter 6 presents a high resolution energy measurement framework capable of measuring power fluctuations at the micro-watt level for any LPSoC. This chapter presents an energy consumption model that factors in the cost of using multiple processing elements simultaneously in order to explore conditions under which optimal energy-efficiency can be achieved. This chapter also presents strategies to achieve performance and energy-optimal work distribution across multiple processing elements of the Keystone II.

In order to study energy-efficiency, it is imperative to be able to measure energy consumption. While it is easily made possible on contemporary HPC systems, for e.g. Intel Running Average Power Limit (RAPL) provides CPU counters that report energy consumption, it is rare to find an LPSoC system with built in current sensors and energy measurement support. To address this problem, a novel energy measurement framework is described for LPSoC systems. Its measurement API is designed such that measurement can be triggered at an instruction level directly from the application during runtime similar to measuring process time.

Optimal performance on the Keystone II requires all ARM and DSP cores to be used simultaneously. For a certain work distribution, using all ARM and DSP cores always leads optimal performance compared to using them in isolation. Whether this work distribution also corresponds to optimal energy-efficiency is not clear however. The energy consumption model developed in this chapter outlines empirical conditions under which optimal energy-efficiency is possible based only on relative performance of processing elements and their active and idle power consumption.

This model is shown to be applicable to any heterogeneous system. Experimental measurements are taken across the TI Keystone II alongside two other contemporary LPSoCs, the NVIDIA TK1 and TX1, and conventional HPC systems with Intel Xeon CPUs and NVIDIA GPUs. These measurements are then used to validate the energy model. As predicted by the model in all cases, optimal performance does not always correspond to optimal energy-efficiency for any heterogeneous system. Using this model, it is now possible to decide during runtime whether to use certain processing elements in order to maximize energy efficiency or performance.

Chapter 7 summarizes key contributions of this thesis, critiques aspects of this work while reflecting upon implications for the wider HPC community, and discusses future work.

1.2 Publications

Contributions from work presented in this thesis have been published across multiple peer-reviewed workshops and conferences. A list of publications in reverse chronological order and the chapter associated with its contribution are given below:

- Chapter 5: **Gaurav Mitra**, Jonathan Bohmann, Ian Lintault, Alistair P.Rendell. *Development and application of a hybrid programming environment on an ARM/DSP system for High Performance Computing*, (Submitted to) 32nd IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018

- Chapter 6: **Gaurav Mitra**, Andrew Haigh, Anish Varghese, Luke Angove, Alistair P.Rendell. *Split Wisely: When work partitioning is energy-optimal on heterogeneous hardware*, The 18th IEEE International Conference on High Performance Computing and Communications, HPCC 2016

- Chapter 4: **Gaurav Mitra**, Eric Stotzer, Ajay Jayaraj, Alistair P. Rendell. *Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture*, The 10th International Workshop on OpenMP, IWOMP 2014.

- Chapter 4: Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, **Gaurav Mitra**, Alistair P. Rendell, Ian Lintault. *OpenMP on Low-Power TI Keystone II ARM/DSP System-on-chip*, The 9th International Workshop on OpenMP, IWOMP 2013.

- Chapter 3: **Gaurav Mitra**, Beau Johnston, Alistair P. Rendell, Eric McCreath, Jun Zhou. *Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms*, The 3rd International Workshop on Accelerators and Hybrid Exascale Systems, AsHES Workshop, IPDPS 2013.

Three other publications were accomplished in collaboration with colleagues. However, contributions from these publications, even though being closely related are not presented in this thesis. These are listed below:

- Kristopher Keipert, **Gaurav Mitra**, Vaibhav Sunriyal, Sarom S. Leang, Masha Sosonkina, Alistair P. Rendell, Mark S. Gordon. *Energy-efficient computational chemistry: Comparison of x86 and ARM systems*, Journal of Chemical Theory and Computation, JCTC 2015.

- Anish Varghese, Robert Edwards, **Gaurav Mitra** and Alistair P. Rendell. *Programming the Adapteva Epiphany 64-core Network-on-chip Co-processor*, The International Journal of High Performance Computing Applications, IJHPCA 2015

- Anish Varghese, Robert Edwards, **Gaurav Mitra** and Alistair P. Rendell. *Programming the Adapteva Epiphany 64-core Network-on-chip Co-processor*, The 4th International Workshop on Accelerators and Hybrid Exascale Systems, AsHES Workshop, IPDPS 2014

Chapter 2

Background & Related Work

A System-on-chip (SoC) processor is composed of multiple on-chip compute devices often with disparate Instruction Set Architectures (ISA). Each such device is designed for a specialized task such as running an operating system or rendering graphics. These devices are connected together via a high speed interconnect such as a Network-on-chip (NoC) communication subsystem. SoC processors are used in both mobile computing devices as well as desktop and server class systems. A low-power SoC (LPSoC) such as the TI Keystone II is exclusively used in energy constrained mobile computing devices or in embedded computing applications. Figure 2.1 portrays a typical LPSoC processor.

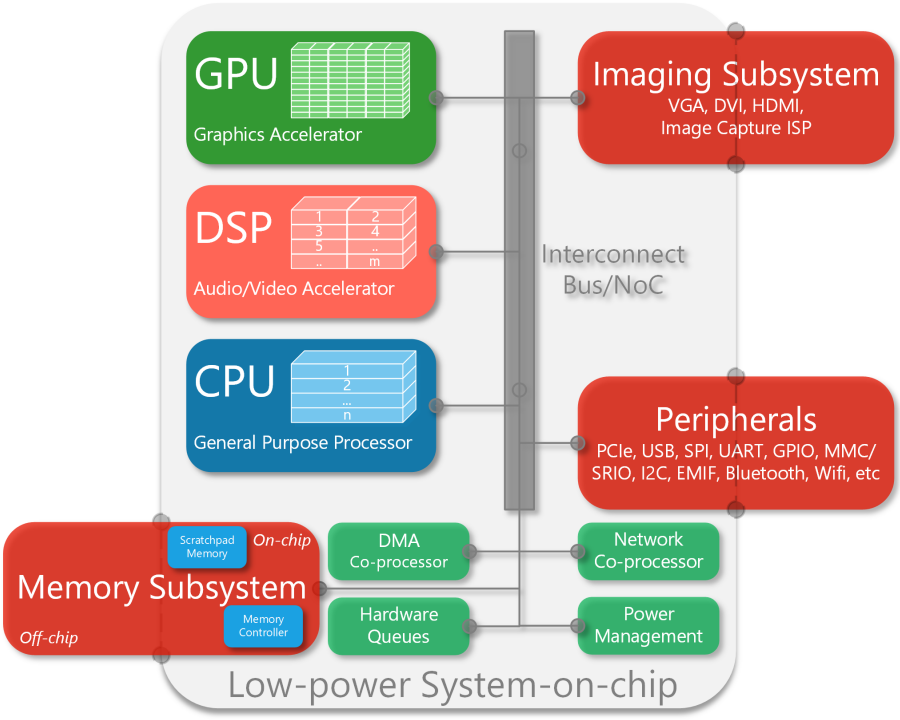


Figure 2.1: A typical Low-power System-on-chip

Two key devices constituting an LPSoC include the CPU and an accelerator such as a Graphics Processing Unit (GPU) or a Digital Signal Processor (DSP). A multi-core CPU with branch prediction, coherent caches and additional floating point units (FPU) is designed to run an operating system and networking stack. A many-core GPU capable of running hundreds of threads simultaneously on multiple data items is well suited to render frames used to display graphics. A multi-core DSP capable of executing Very Long Instruction Word (VLIW) instructions is suited for low-latency real-time tasks such as audio codec processing.

This thesis considers the use of both ARM and DSP cores on the TI Keystone II LPSoC for HPC. An overview of the Keystone architecture and its processing elements is provided in section 2.1. Commercial HPC systems using Keystone SoCs, including the nCore BrownDwarf considered in this thesis, are described in section 2.2.

Methods to improve the use of Single Instruction Multiple Data (SIMD) operations to increase code vectorization on ARM CPU cores are considered in this thesis. Section 2.3 provides a historical overview of SIMD instructions on both ARM and Intel CPUs, categorizes them in terms of functionality.

This thesis considers the use of HPC programming models, OpenMP and OpenCL, to program the Keystone II DSP cores. An overview of both OpenMP and OpenCL is provided in section 2.4 with related work on their use across other LPSoC devices.

Related work is provided in section 2.5.

2.1 The TI Keystone Architecture

The Keystone architecture from Texas Instruments is an innovative platform integrating RISC and C66x DSP cores along with application-specific co-processors and input/output peripherals. It is designed with adequate internal bandwidth enabling non-blocking access to all processing cores, peripherals, co-processors and I/O subsystems.

The first iteration of this architecture, the Keystone I SoC was released in 2010. It consisted of a multi-core C66x DSP and other peripherals. The next iteration, the Keystone II SoC released in 2012, integrated a multi-core ARM Cortex-A15 processor alongside the DSP cores. The Keystone II enabled use of the Linux operating system on the ARM cores, with the DSP cores being exclusively available for compute operations.

2.1.1 The C66x DSP Core

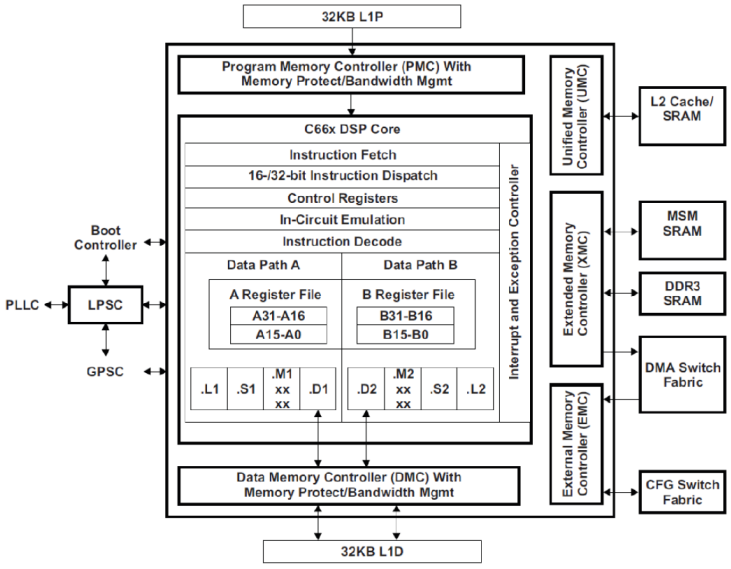


Figure 2.2: The TI C66x DSP core [Texas Instruments Inc., 2010a]

Figure 2.2 presents the architecture of single TI C66x DSP Core. This VLIW DSP core has two data-paths, each capable of executing four instructions per cycle on four functional units named L, S, M, and D. The L and S units perform addition and logical operations. The M unit primarily performs multiplication operations while the D-unit performs load/store and address calculations. The two data-paths appear as an 8-way VLIW machine capable of executing up to eight instructions in each cycle.

The instruction set also includes SIMD instructions allowing vector processing on up to 128-bit vectors. For example, the M unit can perform four single-precision multiplies per cycles, whereas each L and S unit can each perform two single-precision additions per cycle. Together the two data-paths can issue 8 single-precision fused multiply-add FLOPs per cycle. The double-precision capability is about one-fourth of the single-precision FLOPs. This is because only two double-precision fused multiply-add FLOP are possible per cycle compared to eight single-precision multiply-add FLOPs.

There are two general purpose register files each containing 32 registers. Each register is 32-bit wide. Although a particular register file is connected to a particular data-path there is a cross-connect that allows data transfer to the other data-path. The way to achieve high compute efficiency on this architecture is to feed as many instructions as feasible to the functional units without overwhelming the register files and cross-connects.

2.1.2 The Keystone Memory Hierarchy

Keystone SoCs have a Non-Uniform Memory Architecture (NUMA) [Hennessy and Patterson, 2003]. A C66x subsystem can access different memory regions, with accesses to memories that are physically closer to a processor being faster. The different levels of memory and their respective configurations and access times are described below:

- Level-1 program (L1P) and data (L1D): 32KB, 1-cycle access time, configurable as mapped RAM, cache, or a combination of mapped and cached.
- Local-L2: 2-cycle access time, configurable as mapped RAM, cache, or a combination of mapped and cached, and shared between the L1D and L1P caches.
- Multi-core Shared Memory Controller (MSMC) Scratchpad RAM (SRAM): 2-cycle access time, shared memory on-chip.
- DDR3 RAM: multiple gigabytes of off-chip memory with greater than 60-cycle access time.

2.1.3 The Keystone I SoC

The Keystone I *C6678* SoC [Texas Instruments Inc., 2010b] is composed of an octa-core C66x DSP running at 1.25GHz. The DSP cores do not maintain cache coherency amongst them and virtual memory support is absent due to the lack of a Memory Management Unit (MMU).

Figure 2.3 shows the block diagram of this device. It comprises a three-tier NUMA memory hierarchy common to Keystone SoCs with 512KB of L2 memory per DSP core and 4MB of MSMC SRAM.

The Keystone I comes with a set of standard interfaces like PCI express, Serial Rapid I/O (SRIO), multiple Gigabit Ethernet ports as well as a proprietary interface known as the Hyperlink that provides a 50 Gbps point-to-point connectivity. This SoC has one DDR3 memory controller.

The DSP cores are 8-way VLIW capable of eight single-precision fused multiply-add FLOP per cycle. They have a theoretical peak performance of 20 GFLOPS per core and 160 GFLOPS aggregate single-precision GFLOPS (at 1.25 GHz). The double-precision performance of the DSP cores is approximately one fourth single-precision, 40 GFLOPS.

The Keystone I SoC consumes 10 Watts of thermal design power [Ali et al., 2012]. Taking this TDP into consideration, the maximum energy efficiency achievable for the DSP cores would be four double-precision GFLOPS/Watt.

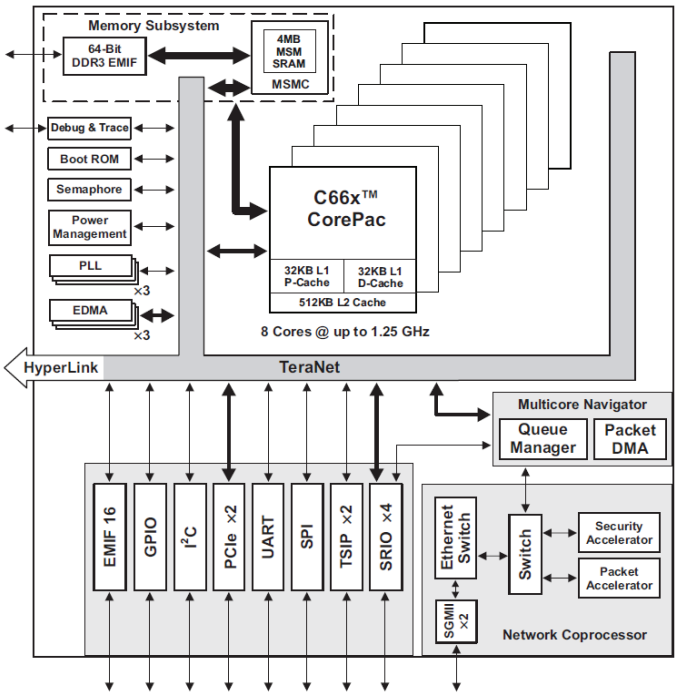


Figure 2.3: The TI Keystone I *C6678* SoC [Texas Instruments Inc., 2010b]

2.1.4 The Keystone II SoC

The Keystone II *66AK2H* SoC [Texas Instruments Inc., 2012] shown in Figure 2.4 is composed of a quad-core ARM Cortex-A15 (up to 1.4 GHz) and an octa-core C66x DSP (up to 1.228 GHz) connected via a 2 Tb/s TeraNet bus.

Compared to Keystone I, DSP memory sizes are increased. The DSP cores still have 32KB of L1D and L1P, however the L2 memory size is increased to 1MB and MSMC SRAM is increased to 6MB. MSMC SRAM is shared by all ARM and DSP cores. Each of the C66x DSP L1 and L2 memories remain configurable and can be partitioned into SRAM and cache as needed. The DSP cores do not maintain cache coherency amongst them, similar to the Keystone I. Virtual memory support remains absent on the DSP cores as they still lack an MMU and do not share the ARM MMU. Section 2.1.1 describes the C66x DSP core.

The ARM cores have 32KB of L1D and L1P cache per core, and share a single 4MB L2 cache. These cores are fully cache coherent. This SoC provides two 72-bit DDR3 interfaces running at up to 1600 Mhz to the DDR3 memory DIMM. The Keystone II has an additional Hyperlink interface compared to the Keystone I.

The Keystone II provides five user-programmable Enhanced DMA 3 (EDMA3) channel controllers capable of asynchronous data transfers to and from DDR

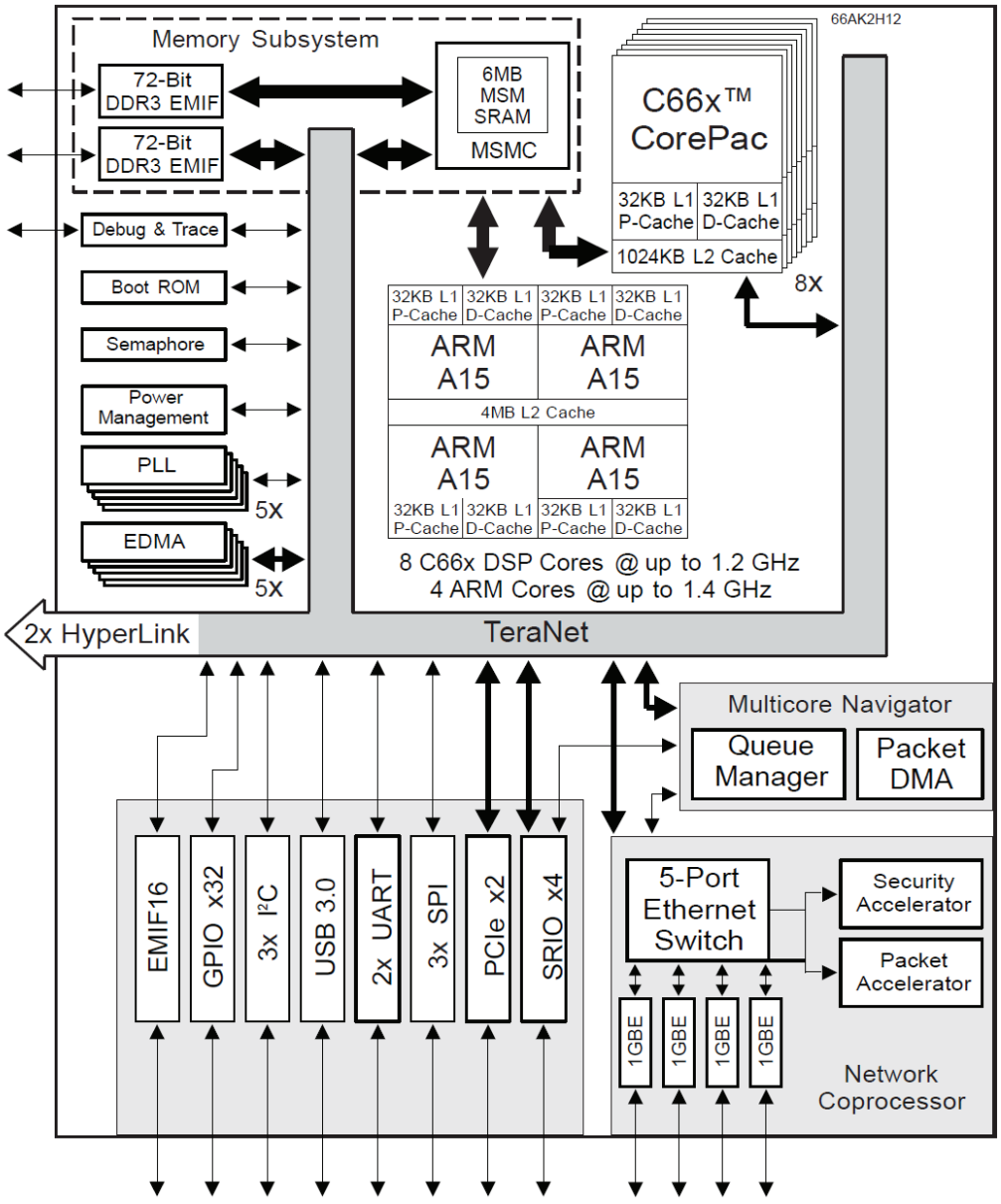


Figure 2.4: TI Keystone II *66AK2H* SoC [Texas Instruments Inc., 2012]

RAM, MSMC SRAM, and L2 memory segments. Each EDMA controller has 64 DMA channels which support transfers triggered by both the user and interrupts/events in the case of chained transfers.

The ARM cores are capable of one double-precision fused multiply-add FLOP per cycle and therefore have a peak performance of 2.4 GFLOPS per core [Rajovic et al., 2013b] and 9.6 GFLOPS aggregate double-precision GFLOPS (at 1.2 GHz). Using the NEON extensions [ARM Limited, 2009], the peak single-precision performance would be approximately four times the double-precision performance, 38.4 GFLOPS.

The DSP cores are 8-way VLIW with two 64-bit loads/stores and four single-precision FLOPS per cycle. They have a theoretical peak performance of 19.6 GFLOPS per core and 157.2 GFLOPS aggregate single-precision GFLOPS (at 1.228 GHz). The double-precision performance of the DSP cores is approximately one-fourth the single-precision performance of 39.3 GFLOPS.

The Keystone II SoC consumes approximately 9-14 Watts of thermal design power (TDP) at 55 degrees C case temperature [tip]. Taking this TDP into consideration, the maximum energy efficiency achievable for the DSP cores would be between 2.86 and 4.37 double-precision GFLOPS/Watt. Counting the ARM cores, this aggregate efficiency ranges between 3.49 and 5.43 double-precision GFLOPS/Watt.

2.2 Commercial HPC systems using Keystone SoCs

Two commercially available HPC systems integrated the Keystone SoCs into their nodes. These are the nCore HPC *BrownDwarf* [nCore HPC, 2014] system, and HPE Proliant m800 server cartridge, part of HPE's *Moonshot* [HP, 2014] project.

2.2.1 The nCore BrownDwarf System

The nCore BrownDwarf system consists of nodes containing both Keystone II and Keystone I SoCs. Each node has one Keystone II connected directly to two Keystone I SoCs via Hyperlink. A maximum of 24 GB of DDR3 memory, with 8GB available to each of the three SoCs, is possible on a node. Figure 2.5(b) portrays a single BrownDwarf node.

Each node is connected to other nodes via three separate interfaces, Serial Rapid-IO (SRIO), 10 gigabit ethernet and 1 gigabit ethernet. SRIO [Zhang et al., 2009] is a non-proprietary, high bandwidth system-level, packet-switched in-

terconnect used for chip-to-chip and board-to-board communications. It has a theoretical bandwidth of up to 20 Gbit/sec.

A commercial off-the-shelf (COTS) PCMIG 3.0 ATCA standard chassis shown in Figure 2.5(a) is used to house BrownDwarf *blades*. Similar to the HPE Moonshot cartridges, different kinds of blades are used in a BrownDwarf system. A compute blade comprises 6 nodes as shown in Figure 2.6. A storage blade provides SSD disk storage. A switch blade is used to route between the three BrownDwarf interconnection networks. Figure 2.6 presents a block diagram comprising different blades used in a BrownDwarf system. A BrownDwarf compute blade has the following specifications:

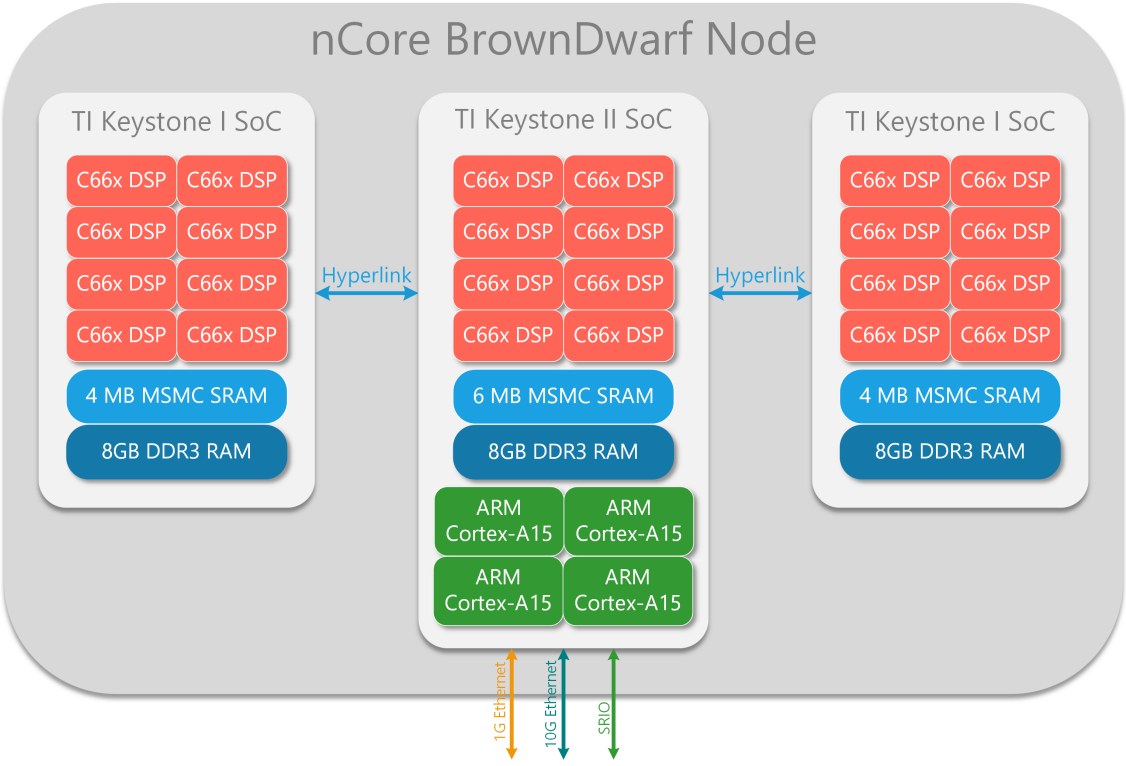
- 6 Processing Units (Compute Nodes or Tiles) with 1 x Keystone II and 2 x Keystone I SoCs per Unit
- 24 x ARM A15 cores @ 1.4GHz
- 144 x C66 DSP cores @ 1.2GHz
- 307.2GB/s Total Memory Bandwidth
- 156GB ECC Memory @ 1600MT/S
- 2TB/s Internal Bus
- 20Gb/s SRIO Compute Fabric (IDT SRIO SoCs)
- 7 x 10Gb Ethernet System Fabric (Broadcom Ethernet SoC)
- 1 x 1Gb Ethernet Front Panel

Any number of these blades can be combined in an ATCA chassis of the correct size to form an HPC cluster. A chassis switch connects all the blades together via the ATCA backplane in a point-to-point configuration.

2.2.1.1 Using the Hyperlink Interconnect

Hyperlink is a low-latency, low-power, 50Gbaud expansion interface designed for co-processor connections to Keystone SoCs [Lin, 2011]. It uses up to four lanes of LVCMOS signaled SERDES, each operating at up to 12.5Gbps, to effect a packet-based transfer protocol with interrupt signaling capability. Within a BrownDwarf node, Hyperlink allows the Keystone II SoC to read from and write to a connected Keystone I SoC's memory. It also allows generation of interrupts in the Keystone I.

(a) BrownDwarf Chassis



(b) BrownDwarf Node

Figure 2.5: nCore BrownDwarf System

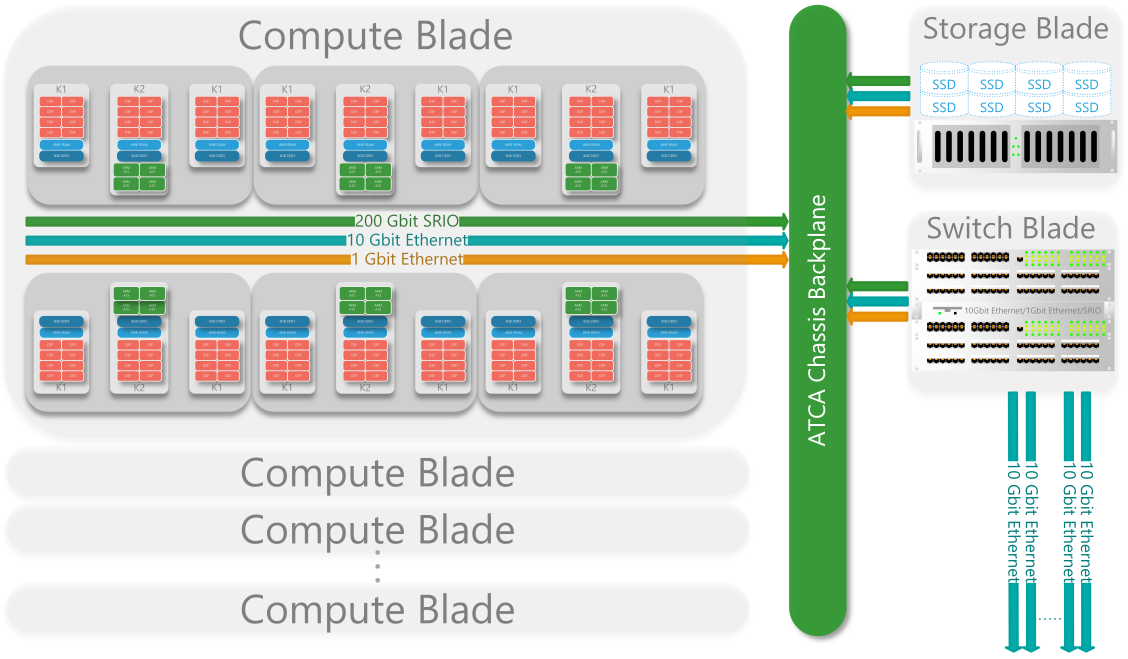


Figure 2.6: BrownDwarf Blade and Chassis Configuration

Hyperlink provides a point-to-point, packet-based, low latency data transfer channel. The Hyperlink protocol, similar to PCiE offers packet read, write and interrupt operations over memory mapped address ranges to a remote device from a local device.

Hyperlink has a Serializer-Deserializer (SerDes) interface but provides a more efficient encoding scheme of 8b9b compared to conventional 8b10b encoding. Between each K2 and K1 SoC there are four lanes of 10 Gbaud Hyperlink. These lanes are configured to run at a stable link rate of 6.25 Gbaud. They offer a combined theoretical bandwidth of 6.25*4*(8/9) Gbps = 22.2 Gbps = 2.78 GB/s of TX and RX between Keystone SoCs.

Hyperlink operates on 64-bit elements. Each word in Hyperlink is 64-bits and is used in two forms, either control words or data words. Each packet contains one to two control words as packet header and then followed by multiple data words.

A transmitted packet across Hyperlink must undergo two stages of address translation. On the originating end, an outgoing (TX) address translation occurs where control information regarding packet destination is overlaid into the address field of the packet. On the receiving end, an incoming (RX) address translation occurs which remaps the incoming address to different memory regions on the device.

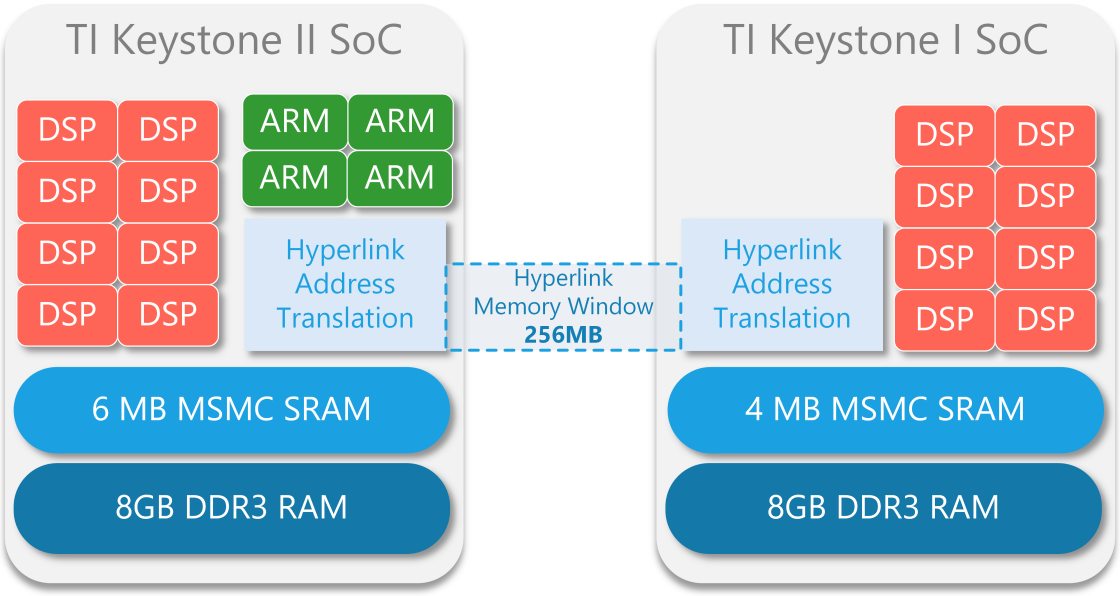


Figure 2.7: Hyperlink Memory Transfer Window

The configuration of Hyperlink memory regions has restrictions. The size of a Hyperlink memory region must be at least 256 B and at most 256 MB. The size must also be a power of two and the region must start on a 64 KB boundary. Up to 64 memory segments are supported regardless of the total memory size. However, there is no restriction on the placement of memory regions across the entire address space. Flexible placement allows different local regions to map to different remote regions of equal size.

The BrownDwarf Hyperlink interface comes pre-configured with 64 memory regions of size 4MB. This provides an addressable window of $64 \times 4 = 256MB$ at a time on a remote node. In order to address a different part of the remote address space, this address window must be moved across. This is depicted in Figure 2.7.

2.2.1.2 Using a 36-bit address space with 32-bit DSP cores

Each SoC on a BrownDwarf node is attached to 8GB of DDR3 memory. Addressing the 8GB requires more than 32-bits. Both Keystone ARM and DSP cores support addressing up to 36-bits. However, 36-bit addressing on the DSP cores is provided by a separate unit on the SoC rather than directly from within the DSP cores.

Since the Keystone DSP cores are 32-bit, any 36-bit address must be trans-

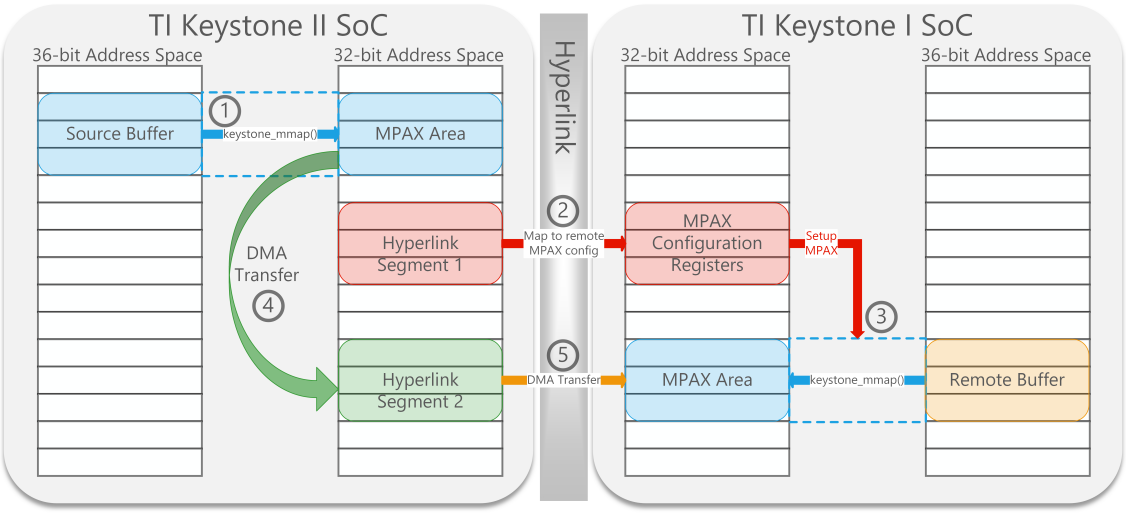


Figure 2.8: Hyperlink Data Transfer using MPAX Address Translation

lated to the corresponding memory region in a 32-bit address space before it may be usable by a Keystone DSP. This translation is provided by the Memory Protection and Address Extension Unit (MPAX) present on a Keystone SoC. MPAX provides registers that can be configured during runtime in order to map different segments of a 36-bit address space to a 32-bit address.

Figure 2.8 describes a data write operation originating from the Keystone II ARM cores using DMA engines to write to a Keystone I DSP DDR memory region on a BrownDwarf node. MPAX is used to perform address translation between 32 and 36 bit addresses in this case. First, a 36-bit source address on the ARM is mapped to a 32-bit source address using MPAX on the Keystone II SoC using a `keystone_mmap()` function call. A memory segment on the ARM is then mapped using Hyperlink to the remote Keystone I SoC's MPAX registers in order to modify them.

The ARM then translates the remote K1's 36-bit address to a 32-bit address by calling the `keystone_mmap()` function again, but this time with the base address configured to be the mapped Hyperlink Segment 1. This modifies the remote MPAX to perform the address translation and designates a memory region in the 32-bit address space to become the destination for the transfer.

This remote region is then mapped via Hyperlink to Segment 2 from the ARM 32-bit address space. DMA co-processors are now used to transfer data from the K2 32-bit mapped source to Hyperlink Segment 2. As soon as data hits this segment, it is transferred to the mapped remote 32-bit address space which in turn is mapped to the remote logical 36-bit destination address.

2.2.2 The HPE Moonshot System

The HPE Moonshot project announced in 2013, comprised a number of different compute *cartridges* with different SoC processors composing each type of cartridge. One such cartridge at the time was the *m800*. It was designed for HPC and was composed of four Keystone II SoCs as shown in Figure 2.9.

These Keystone SoCs were connected together via fast interconnects namely, TI Hyperlink and SRIO as shown in 2.9(b). Each Keystone II SoC had 8 GB of DDR3 memory and was connected externally via gigabit ethernet.

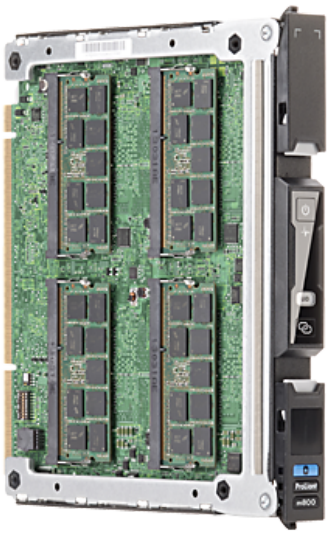
HPE used an in-house manufactured proprietary chassis that contained two types of server cartridges. The first was a compute cartridge, the m800. Recently, HPE updated their compute cartridge family [HPE, 2017] with newer SoCs such as the AMD X2150 Accelerated Processing Unit (APU), composing the m700 cartridge; or Intel Broadwell-DE and Skylake-H Xeon SoCs composing the m510 and m710x respectively. The second type was a storage cartridge providing SSD disk storage.

2.3 Single Instruction Multiple Data Operations

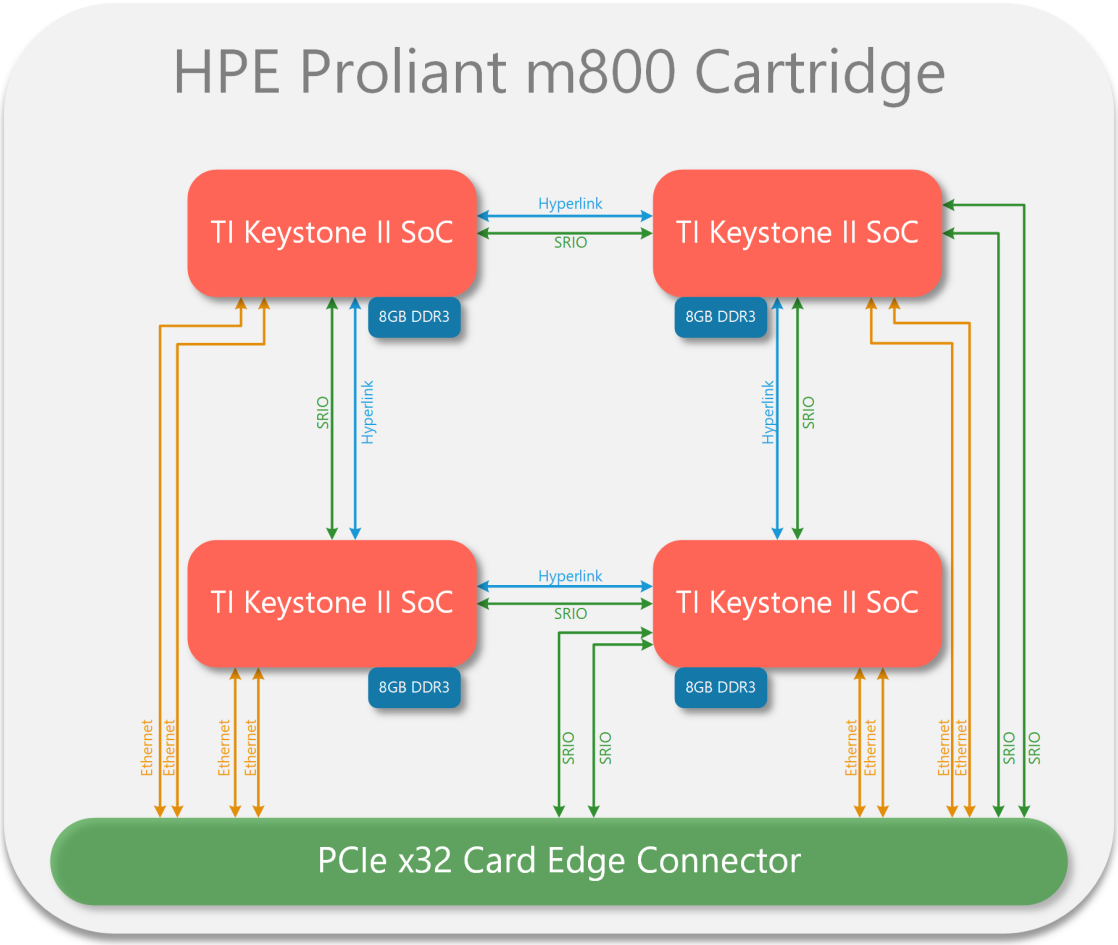
Augmenting a processor with special hardware that is able to apply a Single Instruction to Multiple Data (SIMD) at the same time is an effective method of improving processor performance. It also offers a means of improving the ratio of processor performance to power usage due to reduced and more effective data movement and intrinsically lower instruction counts. This section describes characteristics of SIMD operations, how they are used and compares and contrasts the Intel SSE2 versus the ARMv7 NEON SIMD intrinsic functions.

A SIMD operation acts simultaneously on multiple data elements of the same size and type (in a vector) that have already been fetched into SIMD registers. Compared to sequential or scalar processing, using a SIMD instruction can provide a theoretical speed-up equal to the number of elements that can be operated on by a single SIMD instruction. This speed-up depends on the number of elements of a particular data type that can be packed into a single SIMD register.

A comparison of using scalar and SIMD vector addition is shown in Figure 2.10. In this figure the left-hand side shows the scalar addition of four elements of vector A, with four elements of vector B to produce four elements of vector C. In this process each element of A and B is first read from memory before being added together, and then written back to the memory associated with C. Separate load, add and store instructions are issued for each element of the vector, giving rise to a total of 16 instructions for a vector of length four.



(a) Cartridge [HPE, 2013]



(b) Interconnects

Figure 2.9: HP Moonshot Proliant m800 Cartridge

On the right-hand side of Figure 2.10 the same task is performed using SIMD vector operations. In this approach a single load instruction is used to read four elements of A from memory and similarly for B. A single add instruction sums corresponding elements of A and B, with a final single store instruction used to write four elements of C back to memory. Overall, 16 separate instructions are reduced to four, giving a theoretical overall *speed-up* of four in the context of reduced instruction count. This notion of speed-up is simplified since it does not consider the effects of memory latency and levels of memory hierarchy that a data item needs to traverse before being operated on by a SIMD instruction.

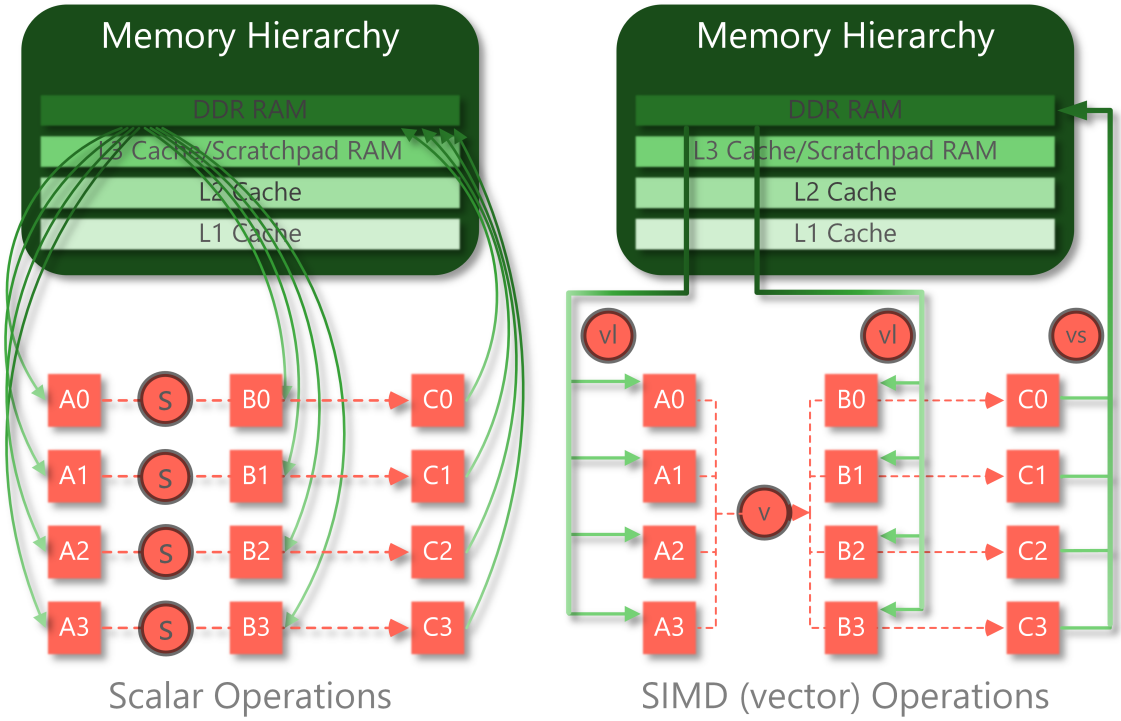


Figure 2.10: Scalar vs. SIMD Vector Addition

The ability to load and store data as fast as possible is as critical as effectively operating on vectors of values using SIMD operations. This is significant given the increasing performance gap between the CPU and main memory. This problem is acknowledged in [Wulf and McKee, 1995] and is known as the *memory wall*. This occurs when modern CPU speeds are vastly faster than resources external to the CPU chip.

Since the *Cray-I* was built in 1976, SIMD *vectorizing* technology has been an integral part of computer development. This is evident today in the existence of Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX)

as part of the instruction set included in the ubiquitous Intel x86 Complex Instruction Set Computer (CISC) processor. SIMD instructions are also present in Reduced Instruction Set Computer (RISC) processors such as the SPARC64 VIIIFX [Maruyama et al., 2010] and the PowerPC A2 [Gschwind, 2012] that are used in the Fujitsu K and IBM BlueGene/Q supercomputer systems.

SIMD operations reduce the total instruction count and enable more efficient data movement which leads to increased energy efficiency [Ibrahim et al., 2009]. For example, in the context of mobile multimedia applications, audio data usually exists in 16-bit data types, while graphics and video data exists in 8-bit data types. Processors used in mobile multimedia are primarily 32-bit such as the ARM Cortex-A8 and A9. Operations on 8 and 16-bit data are performed in 32-bit registers, using roughly the same energy requirements as when processing 32-bit data quantities.

By packing multiple 8 or 16-bit data types into a single 32-bit register and providing SIMD vector operations, the processor can produce multiple results with minimal increase in the required energy use. Further, as many operations are performed at once, the time for the entire computation to complete is shortened, leading to greater reduction in the energy required to produce each result.

SIMD operations are also central to heterogeneous computer systems such as the CellBE processor that was developed by Sony, Toshiba and IBM and used in the PlayStation 3 gaming console [Chen et al., 2007]. These operations are closely related to the Single Instruction Multiple Thread (SIMT) instructions supported on NVIDIA CUDA GPU systems [Kirk, 2007].

In 2009 ARM introduced enhanced floating point capabilities together with NEON SIMD technology into its Cortex-A series of microprocessors [ARM Limited, 2009]. These processors were designed with the smart-phone market in mind. Inclusion of improved floating point and SIMD instructions recognized the rapidly increasing computational demands of modern multimedia applications. In this environment SIMD is attractive since it leads to reduced and more effective data movement and a smaller instruction stream with corresponding power savings. The current range of Cortex-A microprocessors are capable of performing single-precision floating point operations in a SIMD fashion and have recently started supporting double-precision operations in the 64 bit ARM v8-A [ARM Limited, 2012] ISA.

2.3.1 Using SIMD Operations

There are three primary ways of utilizing SIMD instructions on Intel and ARM processors: i) writing low-level assembly code; ii) use of compiler supported

intrinsic functions; and iii) compiler auto-vectorization. There exists another method to influence the generation of SIMD instructions through the use of OpenMP compiler directives. This method is fairly recent and was incorporated in the OpenMP version 4.0 [OpenMP, 2007] standard released in July 2014. Given that compiler support for these directives, a mandatory pre-requisite for evaluating this technique, was non-existent during the course of this study, this method was not evaluated.

Writing low-level assembly code that use SIMD instructions and available registers is often considered the best approach for achieving high performance. However, this method is time consuming and error prone.

Use of compiler supported intrinsic functions provides a higher level of abstraction, with an almost one-to-one mapping to assembly instructions, but without the need to deal with register allocations, instruction scheduling, type checking and call stack maintenance. In this approach intrinsic functions are expanded inline, eliminating function call overhead. It provides the same benefits as inline assembly, improved code readability and less errors [Intel Corporation, 2007]. The penalty is that the overall performance boost becomes dependent on the ability of the compiler to optimize across multiple intrinsic function calls.

Finally, auto-vectorization leaves everything to the compiler, relying on it to locate and automatically vectorize suitable loop structures. Therefore, the quality of the compiler and the ability of the programmer to write code which aids auto-vectorization becomes essential.

2.3.2 NEON and SSE Operations

Prior to execution of SIMD compute operations, data elements of the same size and type must be fetched from memory into specific processor registers. As shown in Figure 2.10, elements A0 to A3 and B0 to B3 must fit into the special SIMD registers before the vector `v` operation can be performed. Vector load operation, `vl` and store operation, `vs` are used to populate the registers.

Intel SSE2 operations work with 128-bit wide XMM registers and 64-bit wide MMX registers. There are eight XMM registers, XMM0-XMM7 and eight MMX registers, MM0-MM7. The newer AVX and AVX2 instruction use 256-bit wide YMM registers.

The ARMv7 architecture [ARM Limited, 2011] defines an extension register set apart from core registers to cater to Advanced SIMD (NEON) and VFPv3 floating point operations. There are thirty-two 64-bit double-word registers, D0-D31, usable by NEON and VFPv3 operations. These registers can also be viewed by NEON operations as sixteen 128-bit quad-word registers, Q0-Q15.

2.3.3 Intrinsic Data Types

Separate data types are defined to represent vectors used as operands in intrinsic functions. These types can only be used as parameters, assignments or return values for intrinsic functions and cannot be used in arithmetic expressions.

In C code, Intel SSE2 uses four types to represent packed data elements [Intel Corporation, 2007]: i) `__m64` can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value; ii) `__m128` can hold four packed single-precision 32-bit floating point values or a scalar single-precision floating point value; iii) `__m128d` can hold two double-precision 64-bit floating point values or a scalar double-precision floating point value; and iv) `__m128i` can hold sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integer values.

ARMv7 NEON data types [ARM Limited, 2009] are more descriptive and are named according to the pattern: $< type >< size > x < number\ of\ lanes > \_t$. Types representing integer data include `int8x8_t`, `int8x16_t`, `int16x4_t`, `int16x8_t`, `int32x2_t`, `int32x4_t`, `int64x1_t` and `int64x2_t`. Unsigned integer data is represented by similar types with `uint` instead of `int` as the $< type >$. There are 2 floating point types, `float32x2_t` and `float32x4_t`, both for single-precision floating point values. NEON also provides 8 and 16-bit polynomial data types, `poly8x8_t`, `poly8x16_t`, `poly16x4_t` and `poly16x8_t`. Further, NEON also provides data types representing arrays of vector types following the pattern: $< type >< size > x < number\ of\ lanes > x < length\ of\ array > \_t$. These types are treated as ordinary C structures. For example, `int16x4x2_t` represents a struct type with parameter `int16x4_t val[2]`. Here, `val[2]` is an array of size 2 holding `int16x4_t` data.

2.3.4 Naming and classification of intrinsic functions

All intrinsic functions follow a naming convention. SSE2 intrinsics are named according to the pattern: $\_mm\_[intrin\_op]\_[suffix]$. Here, *intrin_op* refers to the basic operation of the intrinsic such as `add` or `sub`, while *suffix* denotes the type of data on which the intrinsic operates [Intel Corporation, 2007]. NEON intrinsics follow: $[intrin\_op][flags]\_[type]$. The flag `q` denotes that the intrinsic is using quad-word (128-bit) Q registers [ARM Limited, 2009]. SIMD intrinsic functions can be classified under eight categories:

1. *Data Movement*: Loads and stores of consecutive data elements of the same type and size between SIMD registers and memory. Both SSE2 and NEON support many different load/store combinations for single vectors of each supported data type. NEON further supports load/stores between arrays

of vectors.

2. *Arithmetic*: Addition, subtraction, multiplication, absolute difference, maximum, minimum, and saturation arithmetic are provided by both SSE2 and NEON. Reciprocal and square root operations are also provided by both for floating point data. SSE2 also supports division and averaging instructions for double-precision floating point data. NEON further implements different combinations of these operations such as halving add; multiply accumulate; multiply subtract; halving subtract; absolute difference and accumulate; folding maximum, minimum; operations with a scalar value; and single operand arithmetic.

3. *Logical*: Bitwise AND, OR, NOT operations and their combinations. Both SSE2 and NEON provide AND, OR and XOR operations. Further, SSE2 provides AND NOT operations and NEON provides NOT, bit clear, OR complement and bitwise select operations.

4. *Compare*: Data value comparisons $=, <=, <, >=, >$. Both NEON and SSE2 provide these operations. Further, SSE2 provides the $\neq$ (neq) operation and NEON provides absolute value $>, >=, <, <=$ operations.

5. *Pack, Unpack and Shuffle*: SSE2 and NEON both provide *set* instructions that store data elements into vectors. NEON also supports setting individual items or lanes in vectors, and initializing vectors from bit patterns. SSE2 supports *unpack* and *pack* operations that interleave high or low data elements from input vectors and place them in output vectors. NEON has similar vector extract operations. SSE2 also provides *shuffle* operations that rearrange bits based on a control mask.

6. *Conversion*: Transformations between supported data types. Both SSE2 and NEON provide conversions between floating point and integer data types with saturation arithmetic and low/high element selection within vectors.

7. *Bit Shift*: Often used to perform integer arithmetic operations such as division and multiplication. SSE2 and NEON both have bit shifts by a constant value. NEON further has bit shifts by signed variables; saturation and rounding shifts; and shifts with insert.

8. *Miscellaneous*: Cache specific operations and other applications. SSE2 has casting, bit insert, cache line flush, data prefetch, execution pause and others. NEON has endianness swap, vector reinterpret cast, transpose operations, table lookups and others.

2.4 Programming models for Heterogeneous HPC systems

LPSoC manufacturers have clear preferences over which programming SDK they provide. Several factors dictate this. The most important factor is control over the SDK and the speed at which they can implement and push improvements, bug fixes and new features out to users.

Even though widely accepted and open standards of HPC programming models such as OpenMP and OpenCL have been adopted by most manufacturers, some choose to provide their proprietary ones such as CUDA by NVIDIA. Texas Instruments supports the OpenCL standard and provides a version 1.1 conformant runtime for the Keystone II SoC.

This work considers both OpenMP and OpenCL standards and sections 2.4.1 and 2.4.2 respectively describe the fundamental aspects of these two programming models.

2.4.1 OpenMP

The Open Multi-Processing (OpenMP) specification was designed primarily for shared memory programming across multi-core systems. It is a mature programming standard which was established in 1997 [Dagum and Menon, 1998]. Through the use of compiler directives specified in the OpenMP standard, it is possible to distribute sections of code or *parallel regions* across available processing elements.

The OpenMP 3.0 specification, released in 2008, also supported the notion of lightweight tasks that could be launched simultaneously and would be distributed across available compute cores by the runtime either using a pre-defined or load-balancing scheduling policy.

The OpenMP 4.0 specification, released in 2014, added an accelerator model that enables a programmer to direct execution to heterogeneous processing elements with separate address spaces. Using this model, programmers have the capability to identify the code segments that they want to run on a compute accelerator.

The OpenMP 4.0 *accelerator model* is portable across a variety of different instruction set architectures (ISAs) including GPUs, DSPs and Field Programmable Gate Arrays (FPGA). The model is host-centric with a *host device* and multiple *target devices* of the same type. A *device* is a logical execution engine with some type of local memory. A *device data environment* is a collection of variables that are available to a *device*. Buffers are *mapped* from a host device data environ-

ment to a target device data environment. The model supports both shared and distributed memory systems between host and target devices.

Program execution begins on the initial host device. The `target` construct indicates that the subsequent structured block is executed by a target device. Buffers appearing in `map` clauses are mapped to the device data environment. The `map` clause has a *map-type* which may be specified before the list of variables. The *map-type* is one of `to`, `from`, `tofrom` and `alloc` and is used to optimize the mapping of buffers. Array section syntax is supported on pointer and array variables that appear in `map` clauses to indicate the size of the storage that is mapped.

In Figure 2.11 the `target` construct indicates that the subsequent structured block may execute on a target device. The array sections specified for the buffers *A* and *B* and the buffers *N* and *sum* are mapped to the target device data environment. Use of the `from` map-type indicates that the corresponding buffer *sum* on the target device is not initialized with the value of original *sum* buffer on the host.

Once execution begins on a target device, the iterations of the loop after the `parallel for` construct are executed in parallel by the team of threads executing on the target device. When the `target` region is complete the value of the original buffer *sum* on the host is assigned the value of the corresponding buffer *sum* on the target device and all previously mapped buffers are un-mapped.

The thread on the host device that encountered the `target` construct then continues execution after the `target` region finishes execution.

```
1  double f(double * restrict A, double * restrict B, int N)
2  {
3      double sum;
4      #pragma omp target map(to: A[0:N], B[0:N], N) map(from:sum)
5      {
6          sum = 0.0;
7          #pragma omp parallel for reduction(+:sum)
8          for (int i=0; i<N; i++)
9             sum += A[i] * B[i];
10     }
11     return sum;
12 }
```

Figure 2.11: target construct

Depending on a system's memory hierarchy, a mapped variable might require copies between host and target device memories, or no copies if the host

and target device share physical memory. Even if memory is shared, a pointer translation or memory coherence operation might still be required when mapping a buffer.

The OpenMP 4.0 accelerator model supports hardware configurations with multiple address-spaces. When a buffer in a host data environment is mapped to a corresponding buffer in a device data environment, the model asserts that the host buffer and corresponding buffer *may* share storage. Writes to the device buffer may alter values in the host buffer. Therefore, a program cannot assume that a mapped buffer results in a copy of that buffer being made in device memory.

Other device constructs include the `target data` and `target update` constructs which are used to manage the placement and consistency of variables mapped to device data environments. The `teams` and `distribute` constructs are also added to facilitate a new type of work-sharing pattern that exploits accelerator style loop-level parallelism. These differ from previous loop-level work-sharing using the `for` clause in that the for clause divides work amongst homogeneous processing elements sharing memory, while teams and distribute clauses cater to heterogeneous sets of processing elements within an accelerator which may or may not be sharing memory.

2.4.1.1 The bare-metal OpenMP runtime library for TI C66x DSPs

TI implemented a *bare-metal* OpenMP runtime library for the multi-core C66x DSP using hardware queues on Keystone SoCs. The notion of being bare-metal refers to the absence of an underlying operating system. This runtime itself implements inter-process communication, memory allocation and other necessary mechanisms for program execution that a typical operating system would provide. This runtime is used to execute OpenMP 3.0 regions on Keystone I and II DSP cores in work described in this thesis. This section describes the implementation of this runtime and provides insight into an OpenMP runtime library's internal structure.

Performance of OpenMP based applications depends heavily on the runtime library implementation. Most compilers translate OpenMP into multi-threaded code with calls to a custom runtime library, either via outlining [Brunschen and Brorsson, 2000] or inlining [Liao et al., 2006].

Much of the actual work of assigning computation to different processing elements must be performed dynamically since many execution details are often unknown in advance. Part of the implementation complexity is in ensuring that the presence of OpenMP constructs does not impede sequential optimization in

the compiler. An efficient runtime library which supports thread management, scheduling, synchronization, and fast use of shared memory is essential.

The basic hardware and operating environment of the DSP cores on the Keystone I and II systems presents some special challenges when seeking to support the OpenMP programming model. Notably the shared memory controller in Keystone devices does not maintain coherency between the C66x subsystems such as between two DSP cores, and it is the responsibility of the running program to use synchronization mechanisms and cache control operations to maintain coherent views of the memories.

Without OpenMP, application codes executing across multiple C66x subsystems are required to explicitly manage thread synchronization and cache coherence, and communicate via the shared-L2 and shared-L3 memories. A processor can transfer a data buffer to the local-L2 via a direct memory access (DMA) controller.

The hardware maintains L1D cache coherency with the local-L2 within each DSP core for DMA accesses. Also, the DMA transfer completion event can be used as a synchronization event between the data producer and data consumer. There is no virtual memory management unit (MMU), but a memory protection mechanism protects some shared memory from being accessed by a non-authorized processor.

TI provides a light-weight multi-core task dispatch API called Open Event Machine (OpenEM) which can be used to leverage hardware queues on Keystone SoCs. OpenEM is designed to require minimal memory and CPU cycles [Ins, 2004]. Various types of interactions between cores, such as blocking, communication and synchronization, are implemented by OpenEM. OpenEM also provides a fast, shared, thread-safe memory management system that is used to allocate/deallocate memory in the runtime.

An understanding of the memory model used by OpenMP is fundamental to its implementation on the C66x multi-core DSP. OpenMP specifies a *relaxed consistency* memory model that is close to weak consistency [Hoeflinger and de Supinski, 2005]. In this model threads execute in parallel with a temporary view of shared memory until they reach memory synchronization or *flush* points in the execution flow.

OpenMP has both private and shared variables. A variable may constitute a single data element or an entire buffer. Each thread has its own copy of a private variable that the other threads cannot access. Private variables are located on the stack of each thread of execution. The stack can be placed in any of on-chip local, on-chip shared or off-chip shared memory. There is only one copy of a shared variable, and all threads can access it.

At a flush point, threads are required to write back and invalidate their temporary view of memory. After the memory synchronization point, threads again have a temporary view of memory. Although the C66x provides shared memory, the hardware does not automatically maintain its consistency. It is the responsibility of the OpenMP runtime library to perform the appropriate cache control operations to maintain the consistency of the shared memory when required.

OpenMP requires that threads synchronize their local view of shared variables with the global view at a set of implicit and explicit flush points defined in the OpenMP specification. The runtime performs this synchronization in software. The synchronization steps depend on whether the shared variable is placed in on-chip local memory (L2 SRAM) or on-chip/off-chip shared memory (MSMC SRAM/DDR) as follows:

- Shared variables in on-chip *local* scratch memory(L2 SRAM)
    1. L2 SRAM on a core is accessible to external DSP cores via a global address space
    2. Any updates to L2 scratch by external DSP cores are kept coherent by the memory subsystem
    3. The runtime performs a write-back invalidate of L1 at all flush points.
- Shared variables in on-chip/off-chip shared memory (MSMC SRAM/DDR)
    1. Shared memory regions are marked write-through
    2. The runtime performs cache invalidate operations at all flush points. Since write-through is enabled shared memory has already been updated and there is no need to write-back data.

Consider the implementation of parallel regions. The essential parts of the OpenMP runtime library are implemented using the OpenEM API. For each parallel region, the OpenMP compiler divides the workload into chunks that are assigned to OpenEM tasks (micro-tasks) at runtime.

One of the DSP cores is treated as a master core and the other cores are worker cores. The master core runs the main thread of execution. It is responsible for initializing the OpenMP runtime and starts executing the OpenMP program (main). The worker cores wait in a dispatch loop for OpenEM tasks to show up in a queue. The following steps implement a parallel region's fork-join mechanism:

1. After initialization, worker threads wait in a dispatch loop and check a task queue for micro-task execution notification.

2. The master thread assigns micro-tasks to worker threads as well as assigns a set of them to itself. This is done by posting the micro-tasks to an OpenEM queue. The micro-task description includes a function pointer and a data pointer. It also initializes a shared counter to the number of micro-tasks generated.

3. Worker cores pull micro-tasks out of the OpenEM queue. Upon receipt of the micro-task, each worker thread executes the micro-task specified by the function pointer. The data pointer is passed as an argument to the micro-task.

4. Upon completion of a micro-task, the worker core that executed the micro-task decrements the shared micro-task counter.

5. After the master completes the execution of its own set of micro-tasks, it waits for the shared micro-task counter to reach 0, indicating that all workers have completed their micro-tasks.

The runtime implements three methods of synchronization between threads depending on what is being synchronized:

1. To synchronize master and worker cores during runtime initialization, a fast synchronization mechanism is implemented using coherent shared memory to store a vector. Each core independently sets or clears an element in the vector. Every core can concurrently query the entire vector by using a single 64-bit memory access. As shown in Figure 2.12, this mechanism is based on Lamport's Bakery algorithm [Lamport, 1974]. The buffers are stored in non-cacheable shared memory. The message queue is used at the start of a parallel region, but all other synchronizations are performed using this new mechanism.

2. To synchronize master and worker cores at the end of a parallel region, a shared counter is used.

3. For implicit and explicit OpenMP barriers, the sense reversing barrier shown in Figure 2.13 is used. This has a counter that keeps track of the number of cores participating in the barrier and a sense flag to allow the barrier to be re-used. To avoid coherency overheads, the barrier variable is placed in non-cached memory.

```
1  void sync(char buf0[8], char buf1[8])
2  {
3      int core_id = get_core_id();
4
5      buf1[core_id] = 1;
6      buf0[core_id] = 0;
7      /* wait until all threads have cleared buf0 */
8      while (*(volatile long long *)buf0 != 0) ;
9
10     buf1[core_id] = 0;
11     /* wait until all threads have cleared buf1 */
12     while (*(volatile long long *)buf1 != 0) ;
13
14     /* reset buf0 */
15     buf0[core_id] = 1;
16 }
```

Figure 2.12: Fast synchronization mechanism using coherent shared memory

```
1  void sense_reversing_barrier(Barrier *barrier)
2  {
3      /* To allow re-use, the barrier contains a sense
          variation */
4      char mysense = !barrier->sense;
5
6      if (atomic_decrement(barrier->count) == 1)
7      {
8          /* Last thread resets the sense and count */
9          barrier->count = barrier->size;
10         barrier->sense = !(barrier->sense);
11     }
12     else
13     {
14         /* Modification of sense represents end of the
              barrier */
15         while (mysense != barrier->sense);
16     }
17 }
```

Figure 2.13: Sense reversing barrier

2.4.2 OpenCL

Chapter 4.2.1 will describe the use of the Open Computing Language (OpenCL) version 1.1 runtime library to implement an OpenMP 4.0 runtime.

OpenCL [Khronos, 2011] presents a hierarchical platform model that conforms to several different heterogeneous systems in use today. In this model, a host coordinates execution, transferring data to and from an array of Compute Devices. Each Compute Device is composed of an array of Compute Units, and each Compute Unit is composed of an array of Processing Elements.

This model does not specify exactly what hardware constitutes a compute device. This means that a compute device may be a GPU, such as an NVIDIA K80, or a multi core CPU, such as the Intel Core i7, or other processors such as a DSP or Field Programmable Gate Array (FPGA). The OpenCL platform model is designed to present a uniform view of many different kinds of parallel processors.

OpenCL has a flexible execution model that incorporates both task and data parallelism. Data movements between the host and compute devices, as well as OpenCL tasks, are coordinated via command queues. Command queues provide a general way of specifying relationships between tasks, ensuring that tasks are executed in an order that satisfies the natural dependencies in the computation.

The OpenCL runtime is free to execute tasks in parallel if their dependencies are satisfied, which provides a general-purpose task parallel execution model. Tasks themselves can be comprised of data-parallel kernels, which apply a single function over a range of data elements, in parallel, allowing only restricted synchronization and communication during the execution of a kernel.

OpenCL has a relaxed consistency memory model in which each compute device has a global memory space, which is the largest memory space available to the device, and typically resides in off-chip DRAM. There is also a read-only, limited-size constant memory space, which allows for efficient reuse of read-only parameters in a computation.

Each compute unit on a device has local memory, which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Local memory can be read and written by any work-item in a work-group, and thus allows for local communication between work-groups. Additionally, attached to each processing element is a private memory, which is typically not used directly by programmers, but is used to hold data for each work-item that does not fit in the processing elements registers.

Programmers have control over each memory section and specify exactly

which data segments move in and out of each memory section and how computer kernels operate on these data segments. Programmers also have explicit control over thread execution and can specify the exact sections of data to be worked on by each thread based on unique id's allotted to each thread.

When an OpenCL program is executed, a series of API calls configure the system for execution, an embedded Just In Time compiler (JIT) compiles the OpenCL code, and the runtime asynchronously coordinates execution between parallel kernels.

While it is possible to simply use OpenCL for application codes targeting the Keystone II SoC, OpenCL is a low-level library. It requires in-depth knowledge of library APIs along with the accelerator being programmed and its components. For existing application codes, re-writing them with OpenCL would require significant restructuring and investment. In comparison, the advantage of OpenMP is its ease of use and reliance of simple compiler directives. Taking existing application codes and augmenting them with OpenMP directives requires significantly less effort compared to using OpenCL.

2.5 Related Work

This section describes previous work related to different key issues considered in this thesis. Section 2.5.1 surveys work using LPSoC processors in the HPC domain. Section 2.5.2 describes work relating to the use of SIMD instructions to maximize performance. Section 2.5.3 describes the use of OpenMP and similar compiler directive based models to program accelerators. Section 2.5.4 surveys work relating to the collaborative use of different processing elements such as CPUs and accelerators to increase energy-efficiency of workloads. Section 2.5.5 describes work relating to measurement of energy while section 2.5.6 describes previous energy consumption models.

2.5.1 Low-power System-on-chip processors for HPC

Rajovic et al. [2013a] studied the feasibility of using mobile SoCs for HPC. Their performance and energy efficiency results suggest that LPSoC processors are HPC-ready. However, they identified a number of limitations which need to be addressed. Their work describes experiences in designing and deploying HPC cluster prototypes using low power mobile SoCs and details the problems encountered. Limitations included lack of ECC protection in DRAM, lack of high bandwidth I/O interfaces, lack of hardware support for interconnect protocols. All the limitations identified are design decisions, related to features that are not

required for mobile devices such as smartphones, rather than technical limitations. However, these features are essential for the HPC domain. The authors conclude that if these missing features are included, it is possible that such SoCs may be featured in future HPC systems.

Grasso et al. [2014] analyzed the performance and energy advantages of embedded GPUs, in particular the ARM Mali T-604 GPU integrated into the Samsung Exynos 5250 SoC, for HPC. This GPU achieves a speed-up of 8.7 times over a single Cortex-A15 core, while consuming only 32% of the energy.

Rajovic et al. [2014] built the first large-scale HPC cluster using ARM multicore chips, with energy efficiency as the main focus. The prototype cluster achieves 120 MFLOPS/W using only ARM cores and was competitive with AMD Opteron 6128 and Intel Xeon X5660-based systems. The compute chip used in this work was the NVIDIA Tegra2 SoC, with a dual-core ARM Cortex-A9. The GPU in this SoC, however, did not support programming models such as CUDA or OpenCL and was therefore not used for computation. Simulations were carried out to project the energy efficiency of the cluster if they were built using a large number of higher-performance ARM cores. Their results projected a 16-node cluster configured with ARM Cortex-A15 processors would be competitive with an Intel Sandybridge system and other GPU-accelerated heterogeneous systems in the Green500 list.

Varghese et al. [2014] evaluated Adapteva's Epiphany NoC. It is a low power many-core accelerator with 64 RISC cores with a thermal design power (TDP) consumption of only 2 Watts. It is theoretically capable of delivering 70 GFLOPS/W of energy efficiency. In this work, two applications were implemented to run on the Epiphany NoC accelerator, a 5-point star-shaped heat stencil and single-precision matrix multiplication (SGEMM). Performance was measured at 65.2 GFLOPS for heat stencil and 65.3 GFLOPS for SGEMM. While power was not measured in this work, an energy efficiency of 32.65 GFLOPS/Watt is projected given the TDP of the Epiphany. A drawback of the Epiphany was the absence of double-precision instructions which prevented any double-precision application evaluation.

A crucial aspect of HPC programming on LPSoC systems is data sharing between CPU and accelerator and the availability of a shared address space. Landaverde et al. [2014] investigated the NVIDIA Unified Memory Access (UMA) programming model for sharing memory between CPUs and NVIDIA GPUs. They evaluated UMA performance and found that beyond on-demand data transfers to the CPU, the GPU is able to request subsets of data on demand, allowing UMA to outperform full data transfer methods in many cases. Their results show that the performance of UMA varies significantly based on the

memory access patterns and can perform better than a non-UMA implementation. However, UMA has limited utility due to its high overhead and marginal reduction in code complexity.

Previous work undertaken to evaluate the performance of the NVIDIA Jetson TK1 LPSoC include the work by Fu et al. [2015] which studied the feasibility of running micro-magnetic simulations on mobile GPUs. They were able to achieve higher performance than a desktop CPU system. Although slower than a desktop GPU, the Jetson TK1 achieved higher energy efficiency. Fatica and Phillips [2014] performed synthetic aperture radar imaging on the Jetson TK1 platform, achieving higher performance while consuming less power than conventional HPC systems.

2.5.2 Use of SIMD instructions to improve performance

Several works address performance improvement of audio/video signal processing using NEON compiler intrinsics on ARM platforms, and SSE or AVX intrinsics on Intel platforms.

Infinite Impulse Response (IIR) filters, commonly used as low/high pass filters in audio/video processing, have been accelerated up to 2× using NEON Bentmar Holgersson [2012] and from 1.5× to 4.5× using SSE Kutil [2008].

Video encoding/decoding is improved in multiple works using NEON. These include AVS decoding up to 2.11× [Wan et al., 2012], H.264 decoding up to 1.6× [Rintaluoma, 2009; Rintaluoma and Silven, 2010], and MPEG-4 encoding and decoding up to 2× [Rintaluoma, 2009; Rintaluoma and Silven, 2010].

The Fast Fourier Transform (FFT), a core algorithm for many audio/video and image processing applications was optimized to used SSE, AVX and NEON instructions to create an alternative implementation, SFFT [Blake, 2012]. It was benchmarked against popular FFT implementations such as FFTW and SPIRAL and was shown to be at least as fast or faster than the state-of-the-art, FFTW [Frigo and Johnson, 1998; Turnes and Romberg, 2010] using compiler intrinsic functions on Intel and ARM platforms.

In an evaluation of AVX instructions using only compiler auto-vectorization, Gepner et al. [2011] demonstrated that AVX delivers between 1.58× and 1.88× improvement over SSE 4.2 in computationally intensive benchmarks such as LINPACK and HPCC.

Jeong et al. [2012] observed significant performance advantages of using inline assembly to program SSE 4.2 and AVX instructions compared to compiler auto-vectorized code. Use of compiler intrinsic functions was not considered in this study.

Heinecke and Pflüger [2011] and Heinecke et al. [2012] used data mining applications to compare single-precision and double-precision floating point performance improvements from SSE and AVX instructions using only compiler auto-vectorization. It was shown that AVX intrinsics performed at least 1.63× better than SSE for single-precision computations and at least 1.71× better for double-precision.

Pulli et al. [2012] optimized several computer vision routines using NEON compiler intrinsic functions and benchmarked them on an NVIDIA Tegra 3 device. The objective of this work was to measure comparative improvements offered by NEON versus those offered by OpenGL shading language using the on-chip GPU on the Tegra 3 processor. Impressive results were demonstrated including speed-ups of 23× for median blur, 4.6× for Gaussian blur, 9.5× for color conversion, 3.1× for sobel filtering, 1.6× for canny edge detection, and 7.6× for image resizing.

Apart from SFFT [Blake, 2012], no other work was found comparing performance improvements of SIMD intrinsic functions across Intel and ARM platforms. Additionally, no other comparative analysis of the effect of memory latency on performance of SIMD operations on Intel platforms was found.

Maleki et al. [2011] previously showed that state-of-the-art compilers were able to vectorize only 45-71% of synthetic benchmarks and 18-30% of real application codes on x86 systems. In particular it was noted that compilers did not perform some critical code transformations necessary for facilitating auto-vectorization. Non-unit stride memory access, data alignment and data dependency transformations were found to be pertinent issues with compiler auto-vectorization. No ARM systems were considered in this study. This thesis analyzes similar issues but concentrates on the advantages of using intrinsic functions to perform SIMD operations, and provides benchmark speed-ups compared against those achieved using auto-vectorizing compiler options on the original code across both Intel and ARM systems.

2.5.3 Use of OpenMP on accelerators

Igual et al. [2012] first demonstrated the use of OpenMP to program TI C66x DSP cores for HPC application codes. They achieved 74 GFLOPS across the octa-core C6678 DSP for single-precision matrix multiplication (SGEMM). Ali et al. [2012] extended this work to other level 3 Basic Linear Algebra Subprograms (BLAS). Use of OpenMP to program C66x DSP cores was further demonstrated by Ahmad et al. [2013]; Note et al. [2013]. Chapman et al. [2009] provides an OpenMP runtime using DSP/BIOS for the TI C64x DSP. Jeun and Ha [2007]

outlines a bare-metal implementation of an OpenMP runtime for the Cradle CT3400 multi-core DSP.

IBM provides an OpenMP compiler [Eichenberger et al., 2006] and runtime library for the Cell Broadband Engine. Extensions to OpenMP to support accelerators were introduced in [Ayguade et al., 2009; Cabrera et al., 2009; Ayguadé et al., 2010; Beyer et al., 2011]. Various RTOSs such as SYS/BIOS [Ins, 2010] and DSP/BIOS [Ins, 2004] have been used on the C6678 DSP.

Accelerator models such as OpenACC [Reyes et al., 2012; Lebacki et al., 2012] offer support for offloading work to the attached accelerators/co-processors using compiler directives/pragmas. Intel's MIC architecture offers support for both automatic and compiler-assisted offloading of work from the host to the Xeon Phi co-processor [Leang et al., 2014a]. The PGI accelerator model [Wolfe, 2010] and hiCUDA [Han and Abdelrahman, 2009] have also been use to similar effect.

Methods of allocating buffers in shared memory space are crucial to removing unnecessary data transfers in the OpenMP 4.0 model. Mechanisms to use shared memory are also implemented in NVIDIA CUDA 6 [NVIDIA, 2014b]. CUDA 6 provides the concept of *unified memory* which allows creation of buffers in a shared memory space using `cudaMallocManaged()`. For the new NVIDIA Tegra K1 [NVIDIA, 2014a] SoC which provides four ARM Cortex-A15 cores and a Tegra GPU on-chip, unified memory translates to physically shared memory similar to TI Keystone II. For other systems with discrete host CPU and PCiE connected NVIDIA GPUs, unified memory goes across separate physical memory spaces.

Work presented in § 4.2.1 presents a mapping between OpenMP 4.0 constructs and OpenCL. HOMP [Liao et al., 2013], an early implementation of OpenMP 4.0 for NVIDIA GPUs provides a similar mapping between OpenMP 4.0 and CUDA. The Intel Xeon Phi co-processor is another emerging HPC platform for which OpenMP was one of the first programming models to be implemented [Schmidl et al., 2013; Barker and Bowden, 2013; Cramer et al., 2012; Leang et al., 2014b].

Implementation of compiler directive based accelerator programming for the Xeon Phi was also recently demonstrated by [Newburn et al., 2013].

2.5.4 Collaborative use of multiple devices to increase energy-efficiency

This thesis considers partitioning work across multiple processing elements within a single SoC and also across multiple SoCs. Balancing workloads as well as data

transfers across CPUs and DSPs and/or GPUs is critical in order to achieve good performance and potentially increase energy efficiency.

LaKomski et al. [2015] conducted a study on achieving the balance between energy and performance of hybrid computing applications. They show that performance and energy optimization can be conflicting goals and the optimal balance between the two goals depends both on application characteristics and specific implementations.

The E-ADITHE procedure [Garzón et al., 2016] aims to improve the energy efficiency of iterative computation on current heterogeneous processors by automatically balancing work among the selected resources according to their computational power. Garzón et al. [2016] show that resource selection has a strong impact on the energy efficiency of modern processors.

Static load balancing techniques have been employed in many cases where the load balancing algorithm has a priori information about the parallel application and platform [Beaumont et al., 2001; Yang et al., 2010]. Ohshima et al. [2007] propose a technique for the parallel processing of matrix multiplication that uses both CPUs and GPUs in a heterogeneous environment.

COMPASS [Lee et al., 2015] is a framework that generates a parameterized performance model which can be used for predicting the performance of the target application using automated static analysis. Ren and Suda [2012] developed a load sharing method to adjust the workload assignment between the CPU and GPU components in order to optimize the overall power efficiency.

StarPU [Augonnet et al., 2011] is a simple tasking API which provides dynamic scheduling policies and allows programmers to exploit different accelerators with minimal effort. The MAGMA library [Agullo et al., 2009] automatically allocates specific BLAS calls to either the CPU or GPU depending on their suitability. Individual BLAS calls are not split between both the CPU and GPU.

Donfack et al. [2014] propose to balance workloads at the algorithm level by adjusting tile sizes, effectively varying the relative cost of different subroutines, based on the CPU and GPU's theoretical peak performances by including automatically in the iterative computation: (1) selection of resources which reaches an approximated maximum energy efficiency and (2) and it has to be carried out for every combination of processor/software as well.

Static load balancing techniques have been employed in many cases where the load balancing algorithm has a priori information about the parallel application and platform [Beaumont et al., 2001; Yang et al., 2010]. Ohshima et al. [2007] propose a technique for the parallel processing of matrix multiplication that uses both CPUs and GPUs in a heterogeneous environment. They report a 40% decrease in execution time for a large matrix product when compared to

using only the CPU or GPU.

Beaumont et al. [2001] describe a static partitioning technique for matrix multiplication on a CPU/GPU environment which follows a heuristic based approach. Yang et al. [2010] presents an approach to balance the workload distribution across GPUs and CPUs which results in better performance than static partitioning methods. Using their optimizations, the LINPACK benchmark performed $\approx$ 3.3 times faster on a single node of the TianHe-1 than the vendor's library. They report a 40% decrease in execution time for a large matrix product when compared to using only the CPU or GPU.

Dynamic algorithms do not require a priori information about execution but may incur significant communication overhead due to data migration. Papadrakakis et al. [2011] describes a dynamic balancing approach where the work is divided up into tasks and added to a queue, similar to our approach. Both the CPU and GPU are fed with tasks from this queue.

Catalán et al. [2015] introduces a scheduling policy, which dynamically distributes the workload, for asymmetric multicore processors (AMPs), such as the ARM big.LITTLE SoC. They integrate a coarse-grain scheduling policy, which dynamically distributes the workload between the two core types present in the architecture, combined with a static schedule that re-partitions the work among the cores of the same type. Their experimental results show considerable performance acceleration for the BLAS-3 kernels.

Sao et al. [2014] use techniques such as aggregating small computations to increase compute-bound work and asynchronously assigning compute-bound work to the GPU and memory-bound work to the CPU, thereby minimizing CPU-GPU communication and improving overall system utilization, achieving speedups of $\approx$2 $\times$ over a highly scalable MPI code.

Augonnet et al. [2010] presents an auto-tuning performance prediction approach for load balancing between heterogeneous components. Some general purpose runtime supported programming models developed by Linderman et al. [2008] and Diamos and Yalamanchili [2008] are capable of automatically distributing computation across heterogeneous cores to achieve increased energy and performance efficiency.

Similarly, González-Domínguez et al. [2014] are able to achieve better performance with the computationally intensive task of detecting 2-SNP epistatic interactions in Genome studies using a hybrid combination of Pthreads and CUDA, taking advantage of CPU/GPU architectures. The result outperforms previous approaches which use either only GPUs or only CPUs.

None of the above work considers LPSoC systems or ARM based systems or uses on-chip accelerators with shared memory buffers. Moreover, the mod-

els deal primarily with system-wide power whereas the model presented in this thesis deals with power characteristics of individual components as well as system-wide power. None of the work outlined above considers specific system conditions under which an energy-optimal work distribution between processing elements might exist. This is the primary focus of our model.

2.5.5 Measuring Energy Consumption

Different methods have been used for the measurement of power consumption. The Yokogawa WT230 AC power meter is commonly used for this purpose [Rajovic et al., 2014]. This meter is connected to directly measure power drawn from the AC line. However, this is an expensive instrument costing over 3000 USD. The WattsUp Pro meter is an inexpensive alternative which costs around 150 USD and can be used to measure the AC power consumed by the entire system [Tiwari et al., 2012a].

An alternative to AC measurement is to measure the DC supply to the board using a shunt resistor [Becker et al., 2003]. This approach removes the power supply from the measurement chain, but adds a series resistance to the circuit, which will alter the results of the measurement. A small resistor can be used to minimize this effect, however as the resistance decreases, the accuracy of the measurement also decreases.

Bedard et al. [2010] developed PowerMon, an internal power meter that sits between the system power supply and internal components such as motherboard and hard disk, and can make separate voltage and current measurements on each of six DC power rails and reports measurements at a rate of up to fifty samples per second through a USB interface. This is a relatively inexpensive device which can be reliably used to measure power consumption and energy efficiency of traditional computer systems. However, it cannot be used to measure the power of different components in SoC systems.

Energy measurement on large clusters can be made by measuring the energy of individual components such as CPU, GPU and DRAM. Energy usage of CPUs on Intel based systems (starting from Sandy Bridge) can be obtained using the Running Average Power Limit (RAPL) [Intel, 2011]. NVIDIA Management Library (NVML) provides APIs to measure energy consumed by Tesla and Quadro GPUs [NVIDIA, 2012].

2.5.6 Modeling Energy Consumption

Tiwari et al. [2015a, 2012b] model energy usage for predicting component-level

power characteristics of large scale HPC systems. Balaprakash et al. [2013] provides a basis for auto-tuning HPC code by optimizing for time, power, and energy. An empirical study showed that in some settings objectives are strictly correlated and there is a single, ideal decision point while in others, significant trade-offs exist.

Ge et al. [2014] provides an analytical performance and energy model, PEACH, which captures the performance and energy impact of computation distribution and energy-saving scheduling to identify the optimal strategy for best performance or lowest energy consumption. The strategies and models are evaluated on heterogeneous systems with Intel Sandy Bridge host processors and NVIDIA Tesla GPUs.

Komoda et al. [2013] developed empirical models to predict performance and maximum power consumption given the frequencies of the CPU and GPU, and the distribution of workload. Lang and Rünger [2014] introduced an energy model which derives the power usage as a function of CPU clock frequency from regression models.

Tiwari et al. [2012b] uses artificial neural networks to predict component-level power draw and energy usage of certain HPC computational kernels such as matrix multiplication, stencil and LU factorization. The model, which is trained using a very small number of data points in the parameter space, can be used to predict power draw rate and energy usage of the CPUs and DIMMs with minimal error rate.

Tiwari et al. [2015b] presents an analysis of the performance, parallel scalability and energy efficiency of a widely used quantum chemistry code, GAMESS, on a commercially available HP Moonshot 64-bit ARM cluster and an Intel Ivy Bridge System. Their results show great promise for the co-design approach that considers using many low-power cores rather than a small number of heavy-duty powerful cores to deliver an Exaflop system that can operate within the 20MW power envelope. They also present a cross-architecture comparison with an Intel Ivy Bridge system. The performance on one node of Ivy Bridge with 16 cores is found to be matched by a four node run (with 32 cores) on the Moonshot. It was found that doubling the number of cores to complete the execution faster on the ARM cluster leads to better energy efficiency compared to the Ivy Bridge system. It is concluded that advancements in the compiler and the software stacks for the 64-bit ARM architecture will mitigate some of its performance bottlenecks.

Although none of the above consider LPSoC platforms in their evaluations, they do provide detailed energy models for heterogeneous systems, some of which are problem-dependent. These models account for a majority of system

components including processing elements and DRAM. The work presented in this thesis does not recreate existing detailed models, but provides a simplified one which yields a novel problem-independent energy optimality heuristic applicable to any heterogeneous system and can be used to select an energy efficient processing environment for data-centers.

Chapter 3

SIMD operations on ARM CPUs

The primary limitation of a contemporary ARM based LPSoC system for HPC is raw absolute floating-point performance. Compared to conventional server grade processors such as Intel CPUs, absolute performance of an ARM based LPSoC is often several orders of magnitude lower. A majority of a modern Intel CPU's peak performance is derived from its SIMD units, through use of SSE and AVX operations. This is especially prominent in Intel's many-core Knights Landing processor, the peak performance of which is almost entirely dependent on the use of the new AVX-512 instruction set available via its SIMD execution units.

Over the last decade, proprietary compilers such as Intel *icc*, along with open source compilers such as GNU *gcc* and *clang* have made significant improvements in the automatic use of SSE and AVX operations through compiler optimizations. In 2012, however, it was not clear how well ARM's equivalent NEON SIMD unit compared in performance to Intel and in particular it was not clear how well compilers were able to automatically use NEON instructions.

Coupled with the use of several variants of low-power DRAM in different ARM platforms, it was also crucial to understand how memory latency affected ARM NEON operations.

The aim of the work presented in this chapter was to experimentally assess:

- The impact of using ARM NEON SIMD operations on absolute performance of ARM CPUs.
- The performance of compiler auto-vectorization compared with the use of hand-tuned compiler intrinsic functions across both ARM and Intel CPUs
- The impact of DDR memory read latency on SIMD operations across ARM and Intel CPUs
- Comparative performance across four generations of ARM CPU cores, the Cortex-A8, A9, A15 and A57

To evaluate the impact of SIMD operations, benchmarks implemented with the Open Computer Vision (OpenCV) library [Bradski and Kaehler, 2008] were used. Benchmarks in OpenCV have wide-ranging ramifications given the rising popularity of machine learning and computer vision. The rapid adoption of LPSoCs in power constrained HPC application scenarios such as scene recognition in autonomous cars has steadily heightened the importance of fundamental image processing operations such as those implemented in OpenCV.

The OpenCV library contains over 500 such operations for computer vision related applications, capable of performing various image processing tasks on most common image formats on both ARM and Intel platforms. Our experiments use five separate OpenCV application kernels with different computational characteristics, executed across thirteen different hardware platforms including nine ARM and four Intel platforms. In order to maintain consistency, the GNU *gcc* compiler was used across all experiments.

To analyze the impact of DDR memory characteristics on SIMD operations, we collected detailed memory latency measurements using the LMbench [McVoy et al., 1996] memory read latency benchmark. Critical to the impact of SIMD operations is the ability to move data into and out of the relevant registers fast enough. For the benchmarks considered here this becomes an issue as the data size increases and it becomes necessary to retrieve data from another level in the memory hierarchy.

Section 3.1 summarizes the OpenCV image processing operations used as benchmarks in this work. Section 3.2 details the various hardware platforms, software environments, experimental configurations and related methodology used across experiments. Section 3.3 reports on experimental results. Section 3.4 provides an associated discussion supported by memory latency measurements. Section 3.5 provides concluding remarks.

3.1 Benchmarks

To assess the impact of SIMD operations on the absolute performance of ARM and Intel systems, six individual benchmarks were used. The first is used to quantify the effect of input data size on SIMD operation performance by measuring data read latency from DDR memory. The other five benchmarks were used to quantify the advantage of using compiler intrinsic functions compared to relying on compiler auto-vectorization. These benchmarks were implemented by augmenting existing OpenCV kernels with ARM NEON optimized code.

Image processing operations with pre-existing implementations of SSE2 in-

trinsic hand-tuned code within two fundamental OpenCV modules, *Core* and *Imgproc* were chosen. Using ARM NEON SIMD compiler intrinsic functions equivalent hand-tuned variants for ARM platforms were implemented. These benchmarks were also designed to be in increasing order of computational complexity. In the following subsections their computational characteristics are described.

3.1.1 Benchmark 1: Measuring effect of data read latency

Data resident in SIMD CPU registers can be operated on by SIMD instructions. However, to bring data into CPU registers from main memory, scalar or vector read operations are required. Performance of such read operations depends heavily on cache organization and alignment of data in memory and this is particularly true for memory read latencies.

A memory read operation issued by the CPU triggers a series of events in the cache hierarchy. A *cacheline* sized chunk of data containing the requested data element is first read into cache from DRAM. Once data is resident in cache, it is then read from cache into CPU registers. At this stage, it is available for SIMD operations to operate on.

Data can be aligned in memory through correct allocation of buffers to memory page boundaries. Alignment of data directly affects memory read latency. This is because unaligned data often requires multiple cacheline fetches of aligned chunks and then combination operations across these chunks to form the unaligned chunk. Although improving spatial locality of data may help, it is not straightforward to reduce memory load latency as hardware features such as cacheline size are responsible.

Measuring the latency of read operations is of critical importance and is the purpose of this benchmark. One can expect the latency of reading individual data elements to be inversely proportional to the performance of SIMD operations. The lower the read latency, the higher the performance. Memory read latency reflects a combination of the latency of the CPU bus or interconnect, the cache, the DRAM DIMM, or memory chip and the controller. Knowing the memory read latency provides insight on how SIMD operations may be bottlenecked on certain platforms given poor spatial locality of data. Memory read latency also depends on the actual instruction stream executed and how write operations are interleaved with read operations.

The `lat_mem_read` utility provided by the *lmbench* version 3 suite allows the measurement of read latency. It takes two arguments, an array size in MB and a stride size. It has two loops traversing through the array. The inner loop

creates a list of pointers for different strides and traverses through them [McVoy and Staelin, 1996]. The output has two columns with the array sizes (floating point values) in the first column and the corresponding read latency time (in nano seconds) for all elements of that array in the second column. In order to ensure a fair architectural comparison between systems, read latencies are reported in cycles rather than nano seconds.

The `lat_mem_read` utility ensures that only memory read latency is measured and no write operations are issued with the data being unmodified. It assumes that all read instructions execute in a single clock cycle. Thus, a measured latency of 0.0 nsec means that the data requested in a read operation issued in one clock cycle is available for use in the next clock cycle, while values higher than 0.0 nsec imply that the processor is stalled waiting for the arrival of the data.

3.1.2 Benchmark 2: Measuring effect of data size

A fundamental image processing operation is data type conversion between floating point and integer format. This is required for further filtering or transformations across image pixels. After desired operations, floating-point values are then converted back to integers for the processed images to be displayed.

Data format conversion is an embarrassingly parallel operation. The purpose of this benchmark was to assess the effect of data size on SIMD operations in the context of this embarrassingly data parallel operation. One of the main bottlenecks for such operations is the rate at which data can be streamed from disk or main memory via the caches into registers before being operated on.

Use of SIMD load and store operations also provides a comprehensive measure on their effect on data stream rate. One can expect SIMD load and store operations to perform significantly better compared to scalar load and store operations. This benchmark provides insight into this issue.

In many cases when performing floating point to integer conversion there is an overflow, and saturation arithmetic is required. OpenCV provides a template cast operation, *saturate_cast* to perform this task. Its implementation had pre-existing code hand-tuned for Intel CPUs with SSE2 intrinsic functions.

For this benchmark, the float to short-integer saturate cast operation was hand-tuned using ARM NEON intrinsics. The pre-existing OpenCV SSE2 implementation and our ARM NEON optimized implementation is shown in Figure 3.1.

Consider first the OpenCV SSE2 code where the image pixels are initially stored in memory as 32-bit floats. These pixels are processed in groups of eight

```
template<> inline short saturate_cast<short>(float v)
{ int iv = cvRound(v); return saturate_cast<short>(iv); }

template<> inline short saturate_cast<short>(int v)
{
   return (short)((unsigned)(v - SHRT_MIN) <= (unsigned)USHRT_MAX ?
   v : v > 0 ? SHRT_MAX : SHRT_MIN);
}

/* OpenCV Intel SSE2 optimized rounding operation*/
CV_INLINE  int  cvRound( double value )
{
    #if (defined _MSC_VER && defined _M_X64) || (defined __GNUC__ && defined __x86_64__
        && defined __SSE2__ && !defined __APPLE__)
    __m128d t = _mm_set_sd( value );
    return _mm_vtsd_si32(t);
    #else
    return (int)(value + (value >= 0 ? 0.5 : -0.5));
    #endif
}

/* OpenCV un-optimized */
for( ; x < size.width; x++ )
    dst[x] = saturate_cast<short>(src[x]);

/* OpenCV Intel SSE2 optimized cast */
for( ; x <= size.width - 8; x += 8 )
{
    __m128 src128 = _mm_loadu_ps (src + x);
    __m128i src_int128 = _mm_cvtps_epi32 (src128);

    src128 = _mm_loadu_ps (src + x + 4);
    __m128i src1_int128 = _mm_cvtps_epi32 (src128);

    src1_int128 = _mm_packs_epi32(src_int128, src1_int128);

    _mm_storeu_si128((__m128i*)(dst + x),src1_int128);
}

/* ARM NEON optimized cast */
for( ; x <= size.width - 8; x += 8 )
{
    float32x4_t src128 = vld1q_f32((const float32_t*)(src + x));
    int32x4_t src_int128 = vcvtq_s32_f32(src128);
    int16x4_t src0_int64 = vqmovn_s32(src_int128);

    src128 = vld1q_f32((const float32_t*)(src + x + 4));
    src_int128 = vcvtq_s32_f32(src128);
    int16x4_t src1_int64 = vqmovn_s32(src_int128);

    int16x8_t res_int128 = vcombine_s16(src0_int64,src1_int64);
    vst1q_s16((int16_t*) dst + x, res_int128);
}
```

Figure 3.1: Hand-tuning OpenCV `saturate_cast` with ARM NEON intrinsic functions

as indicated by the outer `for` loop. The first SSE intrinsic, `mm_loadu_ps( src + x)`, loads four single-precision floats into a single 128-bit SSE register. It then converts these to four signed 32-bit integers using the SSE intrinsic, `mm_cvtps_epi32(src128)`. The process is then repeated for the next four image pixels. At this point there are two 128-bit SSE registers each containing four 32-bit signed integers. The next SSE instruction, `_mm_packs_epi32 (src_int128,src1_int128)`, takes both 128-bit SSE registers, downcasts all 32-bit integers to 16-bit, and packs the eight resulting values into a single 128-bit SSE register. The final SSE intrinsic, `mm_storeu_si128((__m128*)(dst + x),src1_int128)`, copies the result back to main memory.

The ARM NEON code implemented for this benchmark is similar except for the downcast, which is performed in two stages. The first stage processes each 128-bit register separately, casting each of the 32-bit integers to 16-bits and returning a 64-bit result (`int16x4_t src1_int64 = vqmovn_s32(src_int128 )`). The second stage, `int16x8_t res_int128 = vcombine_s16(src0_int64 , src1_int64)`, involves packing the two 64-bit results into a single 128-bit register prior to copying the result back to main memory.

Therefore, the NEON code requires two extra intrinsic function calls compared to SSE code. One can expect an ideal speed-up of between 4-8$\times$ from either SSE2 or NEON implementation since conversions are performed in batches of 4 elements while results are stored back in batches of 8 elements across both implementations. The actual speed-up would depend on the efficiency of the memory subsystem and data read latencies.

3.1.3 Benchmark 3: Binary Image Threshold

A common image processing operation, binary thresholding, which has been widely used for image segmentation [Gonzalez and Woods, 2001] is used for benchmark 3. It requires element-wise comparison between a pixel of interest and a predetermined threshold value. Given an array of pixels and a threshold value, an operation occurs on every pixel depending on whether it is above or below the threshold. The operations vary depending on the type of thresholding. OpenCV provides five different types of image thresholding operations, including binary, to-zero, inverse binary, inverse to-zero and truncate operations. We benchmark binary thresholding as described in § A.1.

In the previous benchmark, the same operation are being applied to each data point. In this benchmark we add a layer of complexity where a data comparison is made before applying the same operation to each data point. This is carried out using bitwise mask operations via SIMD instructions.

3.1.4 Benchmark 4: Gaussian Blur

Image blurring, an operation that convolves an image with a blurring or smoothing filter is used for benchmark 4. The filter is a 2-D array of data which is applied across the image to a halo around each pixel or data point. At each image pixel, the filtered outcome is the weighted sum of itself and the neighborhood pixels, where the weights come from the entry of the filter. We use an anisotropic Gaussian filter with standard deviation set to 1 as described in § A.2.

Adding to the complexity of the previous two benchmarks, this benchmark involves a performing an operation on a data point based on the values of its immediate neighbors or points within its halo. This involves streaming in a window of data for each data point to be evaluated and subsequently increases the data stream rate requirement.

3.1.5 Benchmark 5: Sobel Filter

This benchmark is similar in terms of image processing functionality to benchmark 4 as it implements the same form of spatial convolution as Gaussian Blur. The benchmark differs, however, in terms of computational complexity. Instead of convolving with a 2-D filter, two separable 1-D Sobel Filters are used. This changes the data stream pattern as 1-D windows of data are required for each data point instead of 2-D ones. This is further described in § A.3.

3.1.6 Benchmark 6: Edge Detection

After applying a 2-D Sobel filter and a binary thresholding operation, pixels with low gradient intensity are removed. This benchmark aggregates the computational complexities of the previous benchmarks and represents more of a real-world scenario where edges in an image require detection.

3.2 Methodology

3.2.1 Experimental Platforms

The key characteristics of the thirteen Intel and ARM platforms used here are shown in Table 6.1. In this table the entries in the cache column should be interpreted as L1 Cache size (Instruction and Data) /L2 Cache size /L3 Cache size.

Four generations of Intel platforms, from *Pineview* to *Ivy Bridge*, supporting SSE2 instructions are evaluated. The Atom D510 was chosen because it was targeted towards the mobile embedded market [Intel Corporation, 2010]. The Core 2 Quad Q9400 platform was chosen as a popular representative of Intel's desktop processors in 2012. The Core i7 and Core i5, although having different micro-architectures, represent Intel's first generation of commodity processors which began supporting the AVX instruction set.

Four generations of ARM CPUs are evaluated; two with ARM-Cortex A8, four platforms with A9, two with A15 and one with A57. The ARM Cortex-A57 CPU is the only 64-bit ARM v8 CPU, while the rest are 32-bit ARM v7. A total of nine ARM platforms including the TI Keystone II were chosen based on their support for NEON instructions and the ability for them to be programmed using readily available tools.

The first three ARM platforms were Android smart-phones: i) Samsung Exynos 3110 processor part of the Samsung Nexus S; ii) Texas Instruments OMAP 4460, part of the Samsung Galaxy Nexus; and iii) Exynos 4412 quad core, part of the Samsung Galaxy S3. The next six ran different Linux distributions: iv) Texas Instruments DaVinci processor [Texas Instruments, 2011], running Angstrom Linux; v) Exynos 4412, part of the ODROID-X development platform [ODR, 2013] running Linaro-Ubuntu Linux; vi) NVIDIA Tegra 3, part of the CUDA on ARM Development Kit running Ubuntu Linux; vii) TI Keystone II *Hawking* processor, part of the TI Keystone II Evaluation Module running Arago Linux; viii) NVIDIA Tegra K1, part of the Jetson TK1 system running Ubuntu Linux; and ix) the only 64-bit ARM v8 platform being evaluated, the NVIDIA Tegra X1, part of the Jetson TX1 system running Ubuntu Linux.

3.2.2 Software Configuration

For the OpenCV benchmarks, a test harness for all Intel and ARM platforms was written using C++. For the smart-phones we used OpenCV4Android [OCV] and the Android Native Development Kit [NDK] to write the test harness.

OpenCV 2.4.2 was used, compiled on all platforms using the CMake cross compilation toolchain for single thread execution. The NEON specific cmake toolchains provided from *opencv.org*, were used to compile OpenCV4Android.

All NEON optimization via intrinsic functions were made directly within OpenCV *core* and *imgproc* modules in an analogous fashion to their SSE2 counterparts. These optimizations were turned on and off using the OpenCV function `cv::setUseOptimized(bool onoff)` with the benchmarks labelled accordingly.

ID	PROCESSOR	CODENAME	Launched	Threads/Cores/GHz	Cache L1/L2/L3 (KB)	Memory	SIMD Extensions
				INTEL			
A	Intel Atom D510	Pineview	Q1'10	4/2/1.66	32(I),24(D)/1024/ No L3	4GB DDR2	SSE2/SSE3
B	Intel Core 2 Quad Q9400	YorkField	Q3'08	4/4/2.66	32(I,D)/3072/ No L3	8GB DDR3	SSE*
C	Intel Core i7 2820QM	Sandy Bridge	Q1'11	8/4/2.3	32(I,D)/256/8192	8GB DDR3	SSE*/AVX
D	Intel Core i5 3360M	Ivy Bridge	Q2'12	4/2/2.8	32(I,D)/256/3072	16GB DDR3	SSE*/AVX
				ARM			
E	TI DM 3730	DaVinci	Q2'10	1/1.ARM Cortex-A8/0.8	32(I,D)/256/ No L3	512MB DDR	VFPv3/NEON
F	Samsung Exynos 3110	Exynos 3 Single	Q1'11	1/1.ARM Cortex-A8/1.0	32(I,D)/512/ No L3	512MB LPDDR	VFPv3/NEON
G	TI OMAP 4460	Omap	Q1'11	2/2.ARM Cortex-A9/1.2	32(I,D)/1024/ No L3	1GB LPDDR2	VFPv3/NEON
H	Samsung Exynos 4412	Exynos 4 Quad	Q1'12	4/4.ARM Cortex-A9/1.4	32(I,D)/1024/ No L3	1GB LPDDR2	VFPv3/NEON
I	Samsung Exynos 4412	ODROID-X	Q2'12	4/4.ARM Cortex-A9/1.4	32(I,D)/1024/ No L3	1GB LPDDR2	VFPv3/NEON
J	NVIDIA Tegra T30	Tegra 3, Kal-El	Q1'11	4/4.ARM Cortex-A9/1.4	32(I,D)/1024/ No L3	2GB DDR3L	VFPv3/NEON
K	TI Keystone II 66AK2HX	Hawking	Q2'12	4/4.ARM Cortex-A15/1.4	32(I,D)/4096/ No L3	2GB DDR3	VFPv3/NEON
L	NVIDIA Jetson TK1	Tegra K1	Q1'14	4/4.ARM Cortex-A15/2.3	32(I,D)/2048/ No L3	2GB LPDDR3	VFPv3/NEON
M	NVIDIA Jetson TX1	Tegra X1	Q3'15	4/4.ARM Cortex-A57/2.2	32(I,D)/2048/ No L3	4GB LPDDR4	VFPv4/NEON

Table 3.1: Platforms used in SIMD benchmarks

GCC 4.6.3 was used to compile OpenCV and the test harness across all Intel and ARM platforms except the Jetson TX1 which being the most recent 64-bit ARM platform, only supported GCC 5.4.0. The Android SDK level 15 and NDK r8b compiler, using GCC 4.6.x, was used for the Android phones. GCC 4.6.3 with all optimizations disabled with `-O0` was used to compile lmbench without any modifications on Intel and ARM platforms (gcc 5.4.0 was used on Jetson TX1). It was important to use the same compiler version across all systems to have a consistent comparison of the effectiveness of its auto-vectorization capabilities.

Use of the gcc compiler ensures consistency across Intel and ARM platforms. To trigger auto-vectorization on the Intel platforms, `-O3 -msse -msse2` compiler options were used in all cases. ARM auto-vectorization flags included combinations of the following: `-mfpu=neon -ftree-vectorize -mtune=cortex-a8 -mtune=cortex-a9/a15 -mfloat-abi=softfp -mfloat-abi=hard -O3`. All code was compiled with debugging symbols enabled via the `-ggdb3` compiler flag. To analyze assembly instructions, we used the `objdump` tool to extract code annotated assembly from compiled `.o` object files. We further ran objdump output through `c++filt` to convert debugging symbols into function names.

3.2.3 Experiment Configuration

The image resolutions chosen for our study are commonly used by modern mobile digital cameras. The smallest resolution is 640×480 or 0.3 Mega-pixels (mpx). Other image sizes were 1280×960 (1mpx), 2560×1920 (5mpx) and 3264×2448 (8mpx). Uncompressed bitmap images with corresponding sizes of 1.2MB (0.3mpx); 4.7MB (1mpx); 19MB (5mpx); 23MB (8mpx) were used for all experiments. We cycled through 5 different images of each resolution 25 times, to obtain an average runtime over 125 (5×25) runs of a benchmark. The average or mean value across all 125 runs was recorded with a standard deviation of less than 5% of the mean. We chose to traverse 5 different images in succession to minimize caching effects. In the following tables *AUTO* implies compiler auto-vectorized code and *HAND* implies use of hand optimized intrinsic functions. Time is measured in seconds and speed-up reflects how much faster the HAND experiments were compared to the corresponding AUTO cases. A high resolution timer with an accuracy greater than 10^{-6} seconds was used in all cases. To counter effects of dynamic frequency scaling, `cpufreq-set` was used to set the scaling governor to "*performance*" for platforms that supported this feature.

3.3 Results & Observations

3.3.1 Benchmark 1: Memory Read Latency

Consider first Figure 3.2(a). This figure reports measured memory read latencies in number of CPU cycles for two Intel platforms, the Core i5 and the Core 2 Quad for a stride size of 16 bytes. We see that for small sizes up to 0.03125MB i.e. 32KB of contiguous data reads, the measured latency is 4 cycles for the Core 2 Quad and 5 cycles for the Core i5. This represents the latency of fetching data from L1 cache.

For larger data sizes up to 2MB, the latencies are measured to be approximately 6 cycles for both platforms. This represents data reading from L2 cache for the Core 2 Quad and L2/L3 cache for the Core i5. The Core i5 has 256 KB L2 and 3 MB L3 cache while the Core 2 Quad has a 3 MB L2 and no L3 cache. The measurements indicate that for sizes greater than 256KB, the Core i5's L3 cache provides similar read latencies compared to its L2 cache. Above 3MB, the read latency jumps to 9 cycles for the Core i5 and 12 cycles for the Core 2 Quad. This represents reading from DDR3 RAM. The approximate size of each level in the memory hierarchy is obtained by observing the array size at the point where a plateau height changes in the graph.

With a stride size of 16 bytes reported in Figure 3.2(a), spatial locality of data items is higher compared to larger stride sizes. Given that the Intel systems have a cache line size of 128 bytes, multiple 16 byte strides can be accommodated in a single cache line. That is, for stride sizes less than the respective cache line size the apparent latency is reduced as not all read operations correspond to cache line misses.

With increasing stride sizes in figures 3.2(b)-3.2(d) the measured latencies increase significantly across all memory levels for both platforms. Note the difference in scale of the figures with increasing stride sizes. The main memory read latencies for the Core 2 Quad vary between 12-240 cycles and between 9-240 cycles for the Core i5. On average, the Core i5 is 2-5$\times$ faster than the Core 2 Quad depending on stride sizes. This is reasonable since higher stride sizes correspond to lower spatial locality and require multiple cache line fetches to bring in the required data to CPU registers. For the stride size of 512 bytes however, it is observed that the gap in memory latency between the Core i5 and the Core 2 Quad for DDR3 memory fetches diminishes. This indicates that the L3 fetching operation that the Core i5 uses has a block size of less than 512 bytes and its usefulness is minimized for larger data sizes.

Figures 3.3(a)-3.3(d) represent measurements for five of the ARM linux plat-

forms each with a cacheline size of 64 bytes. Consider first Figure 3.3(a). The L1 data cache read latencies are around 5 cycles for the two A9 systems, the K2 A15 and the TK1 A15 and 6 cycles for TX1 A57. The TI Hawking A15 and the TK1 A15 systems show a spurious spike in the read latency around the 0.01758 MB array size. As all systems have the same 32KB L1 cache size this is probably due to the fact that the A15 systems use a 2-way set associative L1 cache with a least recently used replacement policy while the A9 systems have a 4-way set associative L1 cache with a pseudo random replacement policy.

Interestingly the L2 and DDR latencies for the Tegra T30 and Exynos 4412 systems are quite different despite the fact that both processors are operating at the same frequency (Figures 3.3(a)-3.3(d)). Large differences are measured for stride values between 16-128. This difference is particularly pronounced for accessing DDR memory using stride 512. In this case, the Tegra T30 takes roughly 50% longer than on the Exynos system.

Comparing the A15 systems, the TK1 and Keystone II have similar L1 and L2 latencies. However, the Keystone II has noticeably better DDR3 latency compared to the TK1.

Considering DDR latencies, the Exynos A9 shows remarkably better performance compared not only to the Tegra T30, but even 2.5× on average better than both the A15 systems. This shows that even with older generation memories in use, a more effective memory controller as used by the Exynos SoC can make a significant difference in performance.

The 64-bit TX1 A57 outperforms all other 32-bit ARM platforms at memory level higher than L1. Given that it is equipped with DDR4 RAM as compared to older generation DDR3 and DDR2 memories in other ARM platforms, this is not a surprise.

Comparing across Intel and ARM platforms, consider latencies with stride size of 16 bytes in figures 3.2(a) and 3.3(a). Both the Intel platforms have comparable latencies on all memory levels to the Exynos 4412 and Jetson TX1. Other ARM platforms have 2× - 3× higher latencies than Intel platforms.

For the highest stride size of 512 bytes, the DDR latencies of the Intel platforms increase by over 15× compared to 16 byte strides. In comparison, ARM platform latencies increase in the order of 5×. The Exynos 4412 and TX1 platforms have 2× lower DDR latencies compared to the Intel platforms.

These measurements indicates that the Intel platforms have been enjoying better cache performance compared to ARM for smaller stride sizes. This is expected since the Intel cache line size of 128 bytes is twice the ARM cache line size of 64 bytes. For higher stride sizes, as the number of required cache fetches normalizes across Intel and ARM platforms, DDR latencies of ARM platforms

are lower compared to Intel platforms.

3.3.2 Benchmark 2: Measuring effect of data size

Results for the float to short-integer conversion benchmark are given in Table 3.2. For the smallest image resolution 640 × 480, the Core i5 has the best absolute time with both SIMD intrinsic HAND and AUTO experiments.

For Intel processors the speed-up obtained with HAND varies from 5.27 for the Atom to just 1.34 for the Core 2 Quad. For the ARM processors the speed-up variation is greater, ranging from 13.88 on the Exynos 3110 smart-phone to 3.29 on the Tegra T30 system.

As the image size increases the number of operations performed scales with the number of pixels. This is reflected in absolute execution times which scale almost linearly with image size.

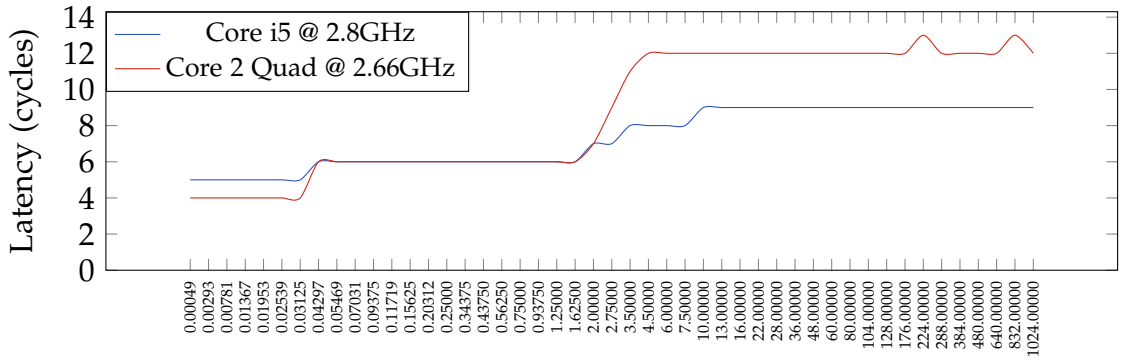
The SIMD speed-ups achieved across all platforms and for all image sizes are plotted in Figure 3.4(a). Results are remarkably similar for all image sizes across the ARM platforms. This is also observed across the Intel platforms.

The speed-up metric compares the performance of SIMD intrinsic code against auto-vectorized code generated from un-optimized source. As this benchmark involves processing four floating point values in one 128-bit wide register, one might expect the maximum speed-up to be a factor of four.

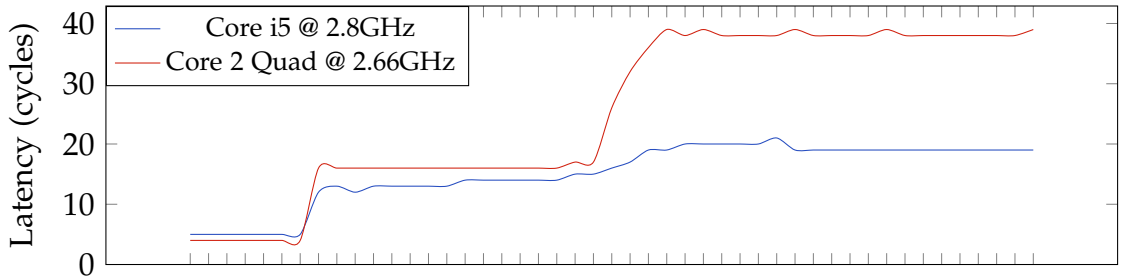
The measured speed-ups vary significantly. On several platforms values greater than four are consistently observed. This is because auto-vectorized code does not always process one floating point value per cycle compared to a SIMD operation's four floating point values per cycle. This is also true for load and store operations. For example, the SSE intrinsic code stores eight short integers in one aligned store instruction while the C code will perform eight separate non-aligned memory store operations.

Considering the ARM Cortex-A9 based SoCs, the 1.4GHz Exynos 4412 in the Galaxy S3 Android smart-phone has the best absolute times on average. This could be attributed to the Exynos 4412 SoC's 3× - 5× lower memory read latency compared to the Tegra T30 system observed in § 3.3.1 along with the lightweight Bionic libc used on the Android OS. For the 1.4GHz ODROID-X and Tegra T30 A9 systems, although the AUTO times are quite similar, the ODROID outperforms the Tegra for the HAND cases. The ODROID shows more than twice as much benefit from using NEON compared to the Tegra T30 as shown in Figure 3.4(a).

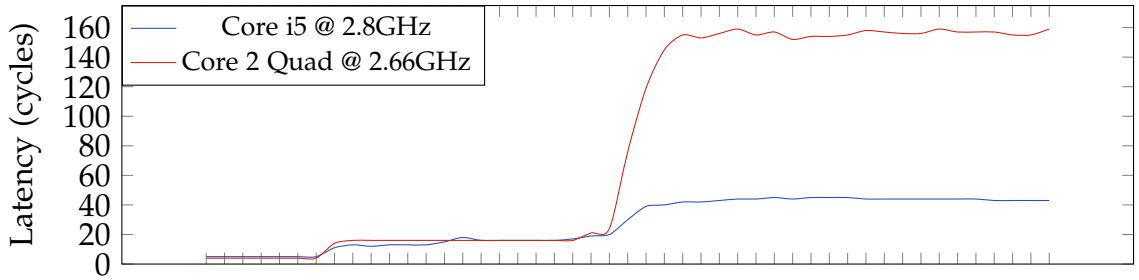
The TI Hawking A15 also running at 1.4GHz performs better than the ODROID. This could be attributed to the improved pipelining present in the A15 compared



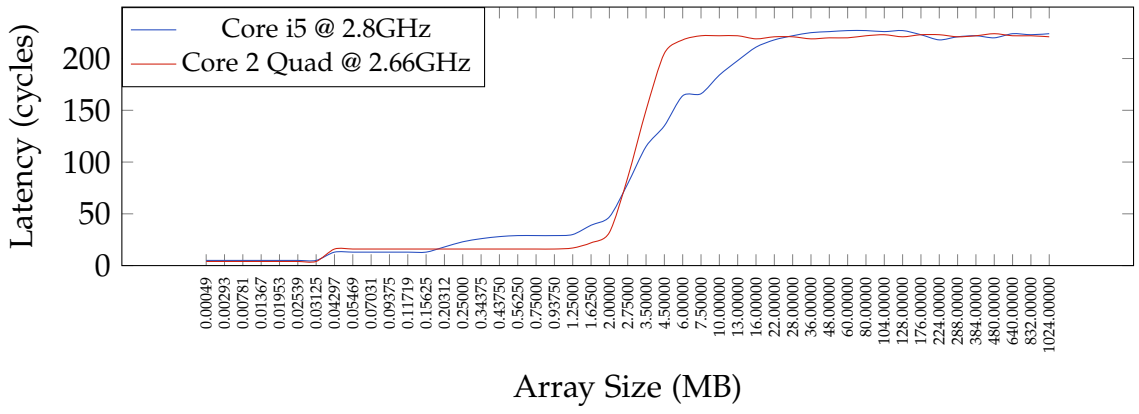
(a) Stride=16



(b) Stride=64

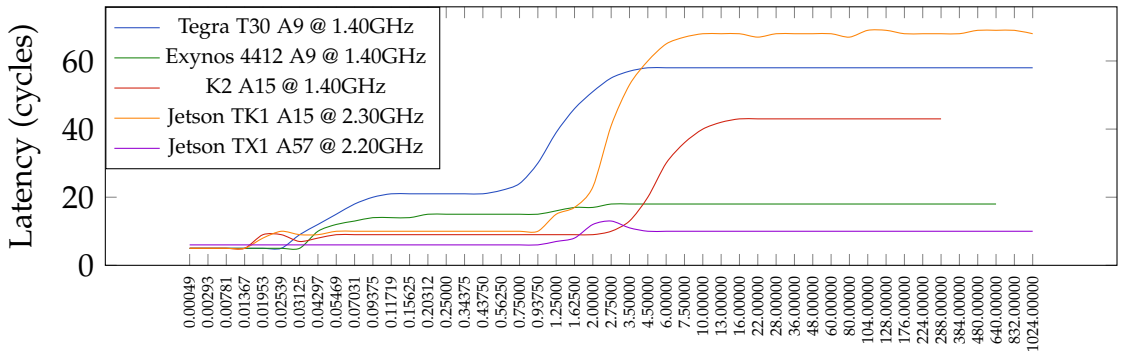


(c) Stride=128

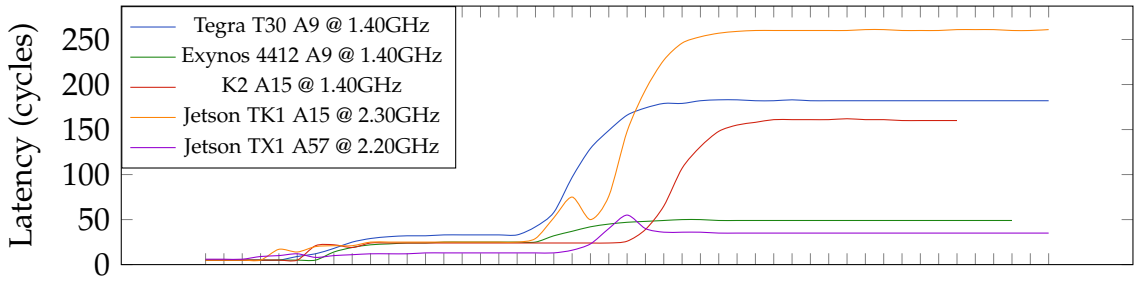


(d) Stride=512

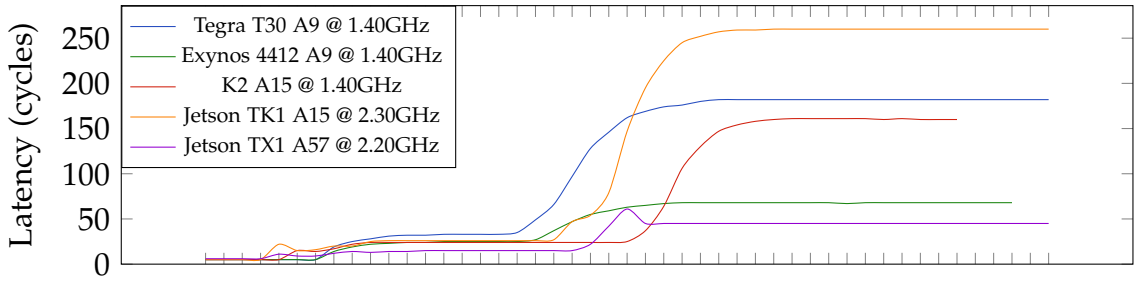
Figure 3.2: Intel Memory Latencies



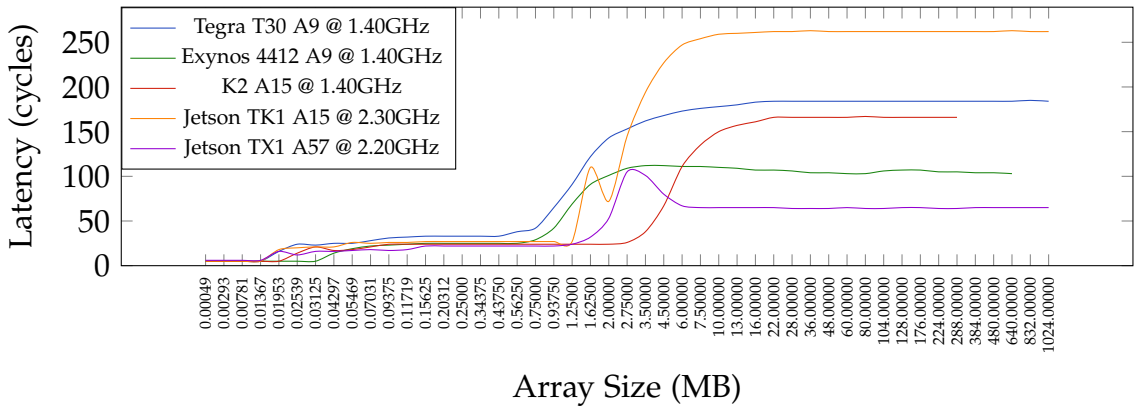
(a) Stride=16



(b) Stride=64



(c) Stride=128



(d) Stride=512

Figure 3.3: ARM Memory Latencies

to the A9. The Jetson TK1 A15, running at a higher clock speed of 2.3GHz has higher absolute performance than the TI K2 A15. However, normalizing w.r.t clock speed reveals that the K2 has similar performance for AUTO cases and marginally higher performance for HAND cases compared to TK1 across all image sizes. This could be attributed to the larger L2 cache present in the K2. The 64-bit TX1 performs better than all other 32-bit ARM platforms.

Comparing ARM SoCs running Android versus those running Linux distributions with similar ARM cores, for e.g the Android based Exynos 3110 Cortex-A8 compared to the Linux distribution based TI DM 3730 Cortex-A8, we observe the AUTO absolute times for Android smart-phones to be significantly better on average than those on Linux platforms. This could again be attributed to customization of gcc 4.6.x and lightweight BIONIC libc libraries used in the Android NDK. HAND times across all ARM platforms except the Tegra T30, maintain consistency when normalized with respect to clock speed and SIMD register width.

3.3.3 Benchmarks 3-6

Absolute execution times and SIMD speed-ups for the large 3264x2448 image with the Binary Image Thresholding, Gaussian Blur, Sobel Filter and Edge Detection benchmarks are given in Table 3.3. Speed-up results for all image sizes and platforms are presented in Figures 3.4(b) to 3.4(e).

The absolute execution times given in Table 3.3 increase progressively from benchmark to benchmark reflecting the steadily increasing complexity. Based on processor clock speed the i5 might be expected to give the shortest execution times, and this is indeed the case.

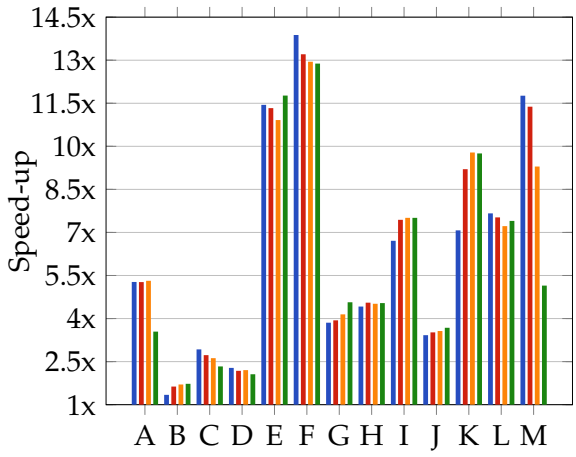
For the Intel platforms, architectural differences mean that the 1.66GHz Atom D510 is about 10× slower than the 2.3GHz Intel Core i7. Specifically, in this case the Intel Atom is an in-order processor while the Intel i7 is an out-of-order processor. This enables the i7 processor to execute non-dependent instructions later in the instruction stream before earlier instructions are complete, while the Atom will stall.

The fastest ARM system in terms of absolute performance is the 64-bit 2.2 GHz NVIDIA TX1. It is 2× to 3× slower compared to the Intel Core i5. The remaining ARM systems are typically 8-15× slower. A fairer comparison is, however, between the Exynos 3110 and the Intel Atom D510 as both these systems are in-order processors. In this case we find the Intel to be 3-10× faster than the ARM system.

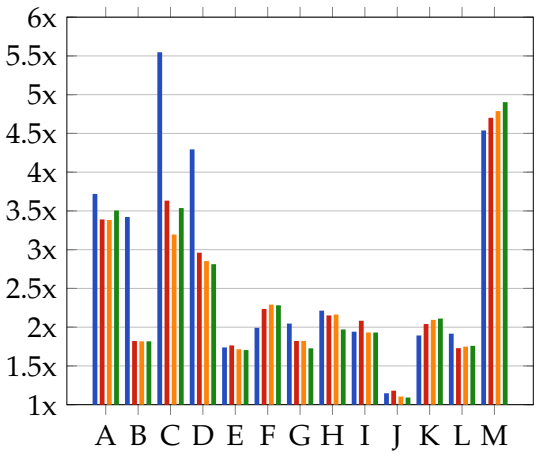
Consider two similar ARM platforms, the Tegra T30 and the ODROID Exynos

SIMD		INTEL (SSE2)				ARM (NEON)							
Image Size	Intrinsic Optimized	**Atom D510**	**Core 2 Q9400**	**Core i7 2820QM**	**Core i5 3360M**	**TI DM 3730**	**Exynos 3110**	**TI OMAP 4460**	**Exynos 4412**	**Odroid-X Ex-4412**	**Tegra T30**	**TI K2**	**NVIDIA TK1**	**NVIDIA TX1**
640x480	AUTO	0.01492	0.00182	0.00122	0.00090	0.20119	0.13215	0.03145	0.02724	0.041274	0.04320	0.02824	0.01632	0.01285
	HAND	0.00283	0.00136	0.00042	0.00040	0.01758	0.00952	0.00816	0.00616	0.00464	0.01311	0.00399	0.00213	0.00109
	Speed-up	5.27	1.34	2.93	2.28	11.44	13.88	3.86	4.42	8.90	3.29	7.07	7.66	11.76
1280x960	AUTO	0.05952	0.00711	0.00483	0.00358	0.80300	0.49577	0.11285	0.10688	0.16465	0.17246	0.11072	0.06511	0.04526
	HAND	0.01129	0.00436	0.00177	0.00164	0.07087	0.03754	0.02866	0.02347	0.01776	0.05205	0.01203	0.00865	0.00397
	Speed-up	5.27	1.63	2.73	2.18	11.33	13.21	3.94	4.55	9.27	3.31	9.20	7.52	11.38
2560x1920	AUTO	0.23770	0.02845	0.01813	0.01417	3.21380	2.01111	0.44328	0.47170	0.65792	0.69020	0.44094	0.26137	0.14965
	HAND	0.04472	0.01670	0.00692	0.00643	0.29443	0.15534	0.10692	0.10447	0.07045	0.20735	0.04506	0.03620	0.01609
	Speed-up	5.32	1.70	2.62	2.20	10.92	12.95	4.15	4.51	9.34	3.32	9.78	7.22	9.30
3264x2448	AUTO	0.43863	0.06392	0.04412	0.03249	5.28033	3.27790	0.92932	0.75601	1.06900	1.12168	0.71727	0.42371	0.29860
	HAND	0.12374	0.03702	0.01892	0.01578	0.44870	0.25445	0.20347	0.16658	0.11392	0.33672	0.07355	0.05724	0.05801
	Speed-up	3.54	1.73	2.33	2.06	11.77	12.88	4.57	4.54	9.38	3.33	9.75	7.40	5.14

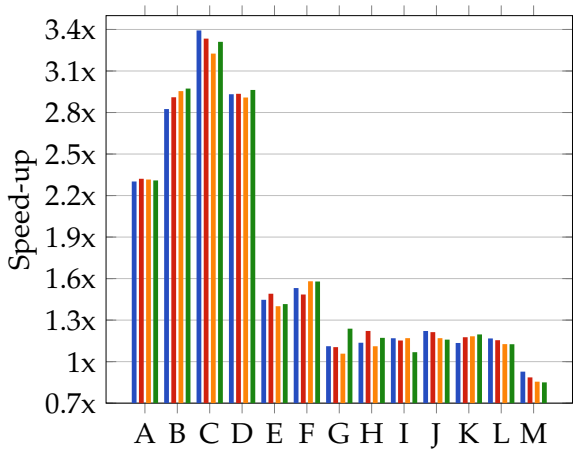
Table 3.2: Time (in seconds) to perform conversion of float to short-integer



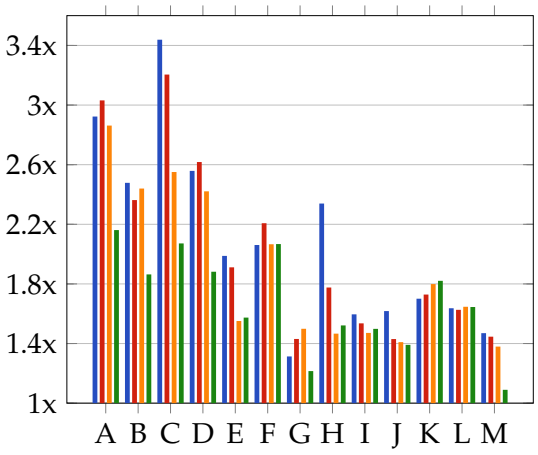
(a) Effect of Data Size



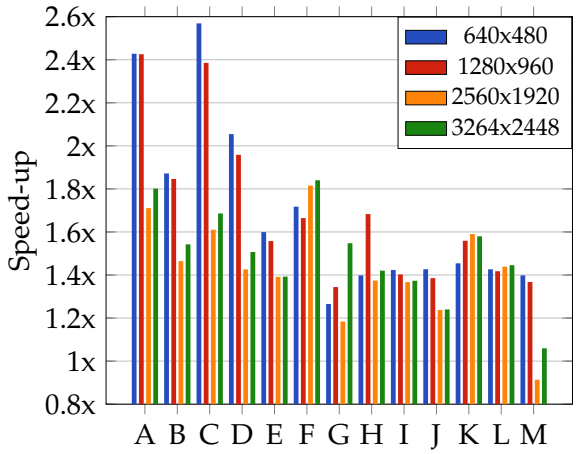
(b) Binary Threshold



(c) Gaussian Blur



(d) Sobel Filter



(e) Edge Detection

A - Intel - Atom D510
B - Intel - Core 2 Quad Q9400
C - Intel - Core i7 2820QM
D - Intel - Core i5 3360M
E - ARM - TI DM 3730
F - ARM - Samsung Exynos 3110
G - ARM - TI OMAP 4460
H - ARM - Samsung Exynos 4412
I - ARM - Samsung Exynos 4412 (ODROID-X)
J - ARM - NVIDIA Tegra T30
K - ARM - TI Keystone II 66AK2HX
L - ARM - NVIDIA Tegra K1
M - ARM - NVIDIA Tegra X1

Figure 3.4: Relative speed-up factors for OpenCV Benchmarks

4412. Both platforms have ARM A9 CPUs running at 1.4GHz. The ODROID consistently outperforms the Tegra T30 in HAND cases, raising questions about what bottlenecks are preventing NEON from performing as well on the Tegra system having the same clock speed and faster DDR3L RAM. Observing the memory read latency measurements from § 3.3.1, it is quite evident that the Tegra T30 memory subsystem has higher latencies compared to all other ARM platform memory levels except the TK1's DDR3. In particular, it has 2-3× higher latency compared to the ODROID Exynos 4412, which leads to the ODROID having higher performance overall.

It is of interest to consider whether an application requires re-tuning with SIMD intrinsic functions to target newer generations of processors with enhanced operations. Since we evaluate both ARMv7 and ARMv8 platforms, it is possible to address this issue. In some cases, for larger image sizes, the ARMv8 TX1 actually shows a decrease in performance for HAND compared to AUTO. This is because compiler intrinsics used in HAND are targeted for ARMv7 32-bit wide SIMD registers and are being executed unchanged on the TX1 ARMv8 64-bit wide registers, while gcc 5.4.0 is able to use auto-vectorization to target ARMv8 specifically on TX1. Moving from one ARM architecture generation to another, it therefore becomes necessary to re-tune previously hand optimized functions to take advantage of increased register size and newer available SIMD instructions.

Compared to the 13× SIMD intrinsic speed-up obtained with benchmark 2, the maximum speed-up observed in Figures 3.4(b)-3.4(e) is about 5.5× across all platforms.

The figures show that in general the benefit of using hand coded SIMD intrinsics over auto-vectorization appears to be slightly greater on the Intel platforms compared with the ARM platforms. The variability of the speed-up also appears to be slightly greater within the Intel platforms compared to within the ARM platforms, even though the same compiler was used in all cases except the TX1.

3.4 Analysis and discussion

3.4.1 Advantage of using compiler intrinsics

To determine why auto-vectorization does not provide similar benefits to direct use of intrinsic functions, we analyzed the assembly instructions for benchmark 2 for both INTEL and ARM platforms. The critical section of the intrinsic optimized Intel SSE2 assembly is presented first in Figure 3.5, followed by the equivalent auto-vectorized assembly.

	SIMD	INTEL (SSE2)				ARM (NEON)								
Image Size	Intrinsic Optimized	**Atom D510**	**Core 2 Q9400**	**Core i7 2820QM**	**Core i5 3360M**	**TI DM 3730**	**Exynos 3110**	**TI OMAP 4460**	**Exynos 4412**	**Odroid-X Ex-4412**	**Tegra T30**	**TI K2**	**NVIDIA TK1**	**NVIDIA TX1**
Binary Threshold	AUTO	0.06688	0.02010	0.01538	0.01075	0.24731	0.17236	0.12912	0.10356	0.11014	0.12152	0.06198	0.04060	0.03225
	HAND	0.01908	0.01106	0.00435	0.00382	0.14508	0.07553	0.07477	0.05255	0.05394	0.11446	0.02936	0.02308	0.00658
	Speed-up	3.51	1.82	3.54	2.81	1.70	2.28	1.73	1.97	2.04	1.06	2.11	1.76	4.90
Gaussian Blur	AUTO	0.46885	0.12089	0.08392	0.06027	1.28726	0.85978	0.78505	0.50677	0.48194	0.56316	0.29507	0.17966	0.14503
	HAND	0.20303	0.04066	0.02535	0.02034	0.90927	0.54455	0.63407	0.43248	0.43966	0.52813	0.24661	0.15959	0.17061
	Speed-up	2.31	2.97	3.31	2.96	1.42	1.58	1.24	1.17	1.10	1.07	1.20	1.13	0.85
Sobel Filter	AUTO	0.60975	0.16437	0.10417	0.06873	2.06342	1.18199	1.16707	0.82602	0.70808	0.90961	0.39731	0.23852	0.18523
	HAND	0.28214	0.08819	0.05028	0.03653	1.31097	0.57174	0.96051	0.54293	0.52677	0.71603	0.21822	0.14499	0.17003
	Speed-up	2.16	1.86	2.07	1.88	1.57	2.07	1.22	1.52	1.34	1.27	1.82	1.65	1.09
Edge Detection	AUTO	0.73730	0.21568	0.15100	0.09142	2.39850	1.36388	1.42742	0.93302	0.85462	1.29594	0.46723	0.30081	0.24199
	HAND	0.40933	0.13984	0.08956	0.06067	1.72260	0.74123	0.92232	0.65695	0.67227	1.07118	0.29578	0.20808	0.22841
	Speed-up	1.80	1.54	1.69	1.51	1.39	1.84	1.55	1.42	1.27	1.21	1.58	1.45	1.06

Table 3.3: Time (in seconds) to perform Binary Thresholding, Gaussian Blur, Sobel Filter and Edge Detection benchmarks on 8mpx (3264x2448) images

```
1  /* Intrinsic Optimized Intel Assembly*/
2  //__m128 src128 = _mm_loadu_ps (src + x);
3  80:movups (%rdx),%xmm0
4  //src128 = _mm_loadu_ps (src + x + 4); (0x10 == 16)
5  movups    0x10(%rdx),%xmm1
6  //__m128i src_int128 = _mm_cvtps_epi32 (src128);
7  cvtps2dq  %xmm0,%xmm0
8  //src+x+8 (0x20 == 32)
9  add   $0x20,%rdx
10 //__m128i src1_int128 = _mm_cvtps_epi32 (src128);
11 cvtps2dq  %xmm1,%xmm1
12 //src1_int128 = _mm_packs_epi32(src_int128, src1_int128);
13 packssdw  %xmm1,%xmm0
14 //_mm_storeu_si128((__m128i*)(dst + x),src1_int128);
15 movdqu  %xmm0,(%r8,%rax,1)
16 //dst = x += 16
17 add   $0x10,%rax
18 //check for terminating condition (x > size.width)
19 cmp   %rbp,%rax
20 //if x <= size.width - 8 loop again
21 jne   80 <cv::cvt32f16s(float const*, unsigned long, unsigned char const*, unsigned
       long, short*, unsigned long, cv::Size_<int>, double*)+0x80>
22
23 /* Auto-vectorized Intel Assembly */
24 //saturate_cast loop
25 //Move scalar single-precision floating-point value between XMM registers or between
       an XMM register and memory
26 d0: movss (%r10,%rax,2),%xmm0
27
28 //Convert packed single-precision floating-point values to packed double-precision
       floating-point values
29 cvtps2pd  %xmm0,%xmm0
30 //Convert scalar double-precision floating-point values to a doubleword integer
31 cvtsd2si  %xmm0,%edx
32
33 //perform conversion from int -> short using OpenCV's rounding cast:
34 //#define  CV_CAST_16S(t) (short)(!(((t)+32768) & ~65535) ? (t) : (t) > 0 ? 32767 :
       -32768)
35 //Load effective address src + 2^15 (16 most significant bits)
36 lea   0x8000(%rdx),%esi
37 mov   %edx,%ecx
38 // if 2^16 >= 16 bit raw value of src, store it (check 32->16 does not overflow)
39 cmp   $0xffff,%esi
40 jbe   f5 <cv::cvt32f16s(float const*, unsigned long, unsigned char const*, unsigned
       long, short*, unsigned long, cv::Size_<int>, double*)+0xf5>
41 test    %edx,%edx
42 mov   %ebx,%ecx
43 cmovle  %r12d,%ecx
44 // store to dest
45 f5: mov %cx,(%r9,%rax,1)
46
47 //loop through the image width (if x < size.width jump to d0)
48 add   $0x2,%rax
49 cmp   %r11,%rax
50 jne   d0 <cv::cvt32f16s(float const*, unsigned long, unsigned char const*, unsigned
       long, short*, unsigned long, cv::Size_<int>, double*)+0xd0>
```

Figure 3.5: Analysis of AUTO vs HAND vectorized Intel assembly code for Benchmark 2

We see in line 3 that `_mm_loadu_ps` maps to `movups` instructions, which conforms with [Intel Corporation, 2012]. In line 7, the `_mm_cvtps_epi32` also maps to the `cvtps2dq` operation which converts a single-precision floating point value to packed double word integers. The first `add` instruction is used to update the `src` pointer to the `(src+x+4)` position, $\texttt{0x20}_{16} = 32_{10}$, this new offset is used for the next load (in the next iteration of the for-loop). In line 13, `_mm_packs_epi32` maps to `packssdw` which packs double words into words using a signed saturation. The result 16 bit $\times$ 16 integer vector is then stored. The last add increments the destination pointer by 16.

Consider the corresponding ARM assembly code in Figure 3.6. Mapping the assembly in the first block of code between line 1 to line 26 to the NEON intrinsic instructions from which they are generated is fairly straightforward. The one interesting observation is that the `vcombine_s16` has been replaced with a `vorr` operation in line 21, which is a bitwise OR activity. This is unexpected as [ARM Limited, 2011] states that this operation maps to a `vmov` operation. Overall eight NEON intrinsics translate into eight NEON assembly instructions. An additional six instructions are required to maintain address offsets and control the loop. Thus, a total of 14 operations are required per eight output pixels. For the auto-vectorized assembly some use of NEON instructions is apparent, but clearly the major issue is that the loop is not running in blocks of eight pixels. As a consequence, many more operations are required per output pixel, explaining the large speed-up that was observed on some of the ARM platforms.

Similarly, for the Intel kernels, the AUTO version requires 14 instructions for every eight output pixels compared to 10 HAND instructions. It is important to recognize, however, that loop optimization techniques such as loop unrolling may have reduced the number of instructions required in the AUTO version.

3.4.2 Effect of memory read latency on SIMD operations

One might expect the memory read latency measurements from § 3.3.1 to bear direct correlations to performance results obtained using SIMD benchmarks across all platforms. Lower read latency should correspond to higher performance. This is indeed the case, and more so for small input data sizes that have high cache residency.

In order to further analyze the effect of memory read latency, the pattern of accessing input data must be evaluated. Benchmarks used in this study are restricted to having a 3×3 window of neighboring pixels in cache for each image pixel being processed. This is true even for the most computationally intensive benchmark 6, Edge Detection. This requires at most three cache lines

```
1  /* Intrinsic Optimized ARM Assembly*/
2  48:   mov r2, r1
3  add.w r0, r9, r3  #x+8
4  adds  r3, #16   #src+x
5  adds  r1, #32   #src+x+4
6
7  //float32x4_t src128= vld1q_f32((const float32_t*)(src + x))
8  vld1.32 {d16-d17}, [r2]!
9  cmp r3, fp
10 //int32x4_t src_int128= vcvtq_s32_f32(src128)
11 vcvt.s32.f32  q8, q8
12 //src128 = vld1q_f32((const float32_t*)(src + x + 4))
13 vld1.32 {d18-d19}, [r2]
14 //src_int128 = vcvtq_s32_f32(src128)
15 vcvt.s32.f32  q9, q9
16 //int16x4_t src0_int64= vqmovn_s32(src_int128)
17 vqmovn.s32  d16, q8
18 //int16x4_t src1_int64= vqmovn_s32(src_int128)
19 vqmovn.s32  d18, q9
20 //int16x8_t res_int128= vcombine_s16(src0_int64,src1_int64)
21 vorr  d17, d18, d18
22 //vst1q_s16((int16_t*) dst + x, res_int128)
23 vst1.16 {d16-d17}, [r0]
24 //iterate through width if (x <= size.width - 8)
25 bne.n 48 <cv::cvt32f16s(float const*, unsigned int, unsigned char const*, unsigned int
       , short*, unsigned int, cv::Size_<int>, double*)+0x48>
26 //end of if (cv::useOptimized())
27
28 /* Auto-vectorized ARM Assembly */
29 // vector load multiple registers
30 8e: vldmia      r6!, {s15}
31 // vector convert float 64 bit to float 32 bit
32 vcvt.f64.f32  d16, s15
33 // copy shortened float back to register r0
34 vmov  r0, r1, d16
35 bl  0 <lrint>
36 //perform conversion from float -> int -> short using OpenCV's rounding cast:
37 //#define  CV_CAST_16S(t) (short)(!(((t)+32768) & ~65535) ? (t) : (t) > 0 ? 32767 :
       -32768)
38 add.w r2, r0, #32768  ; 0x8000
39 uxth  r3, r0
40 cmp   r2, r8
41 bls.n b2 <cv::cvt32f16s(float const*, unsigned int, unsigned char const*, unsigned int
       , short*, unsigned int, cv::Size_<int>, double*)+0xb2>
42
43 cmp r0, #0
44 ite gt
45
46 movgt r3, sl
47 movle.w r3, #32768  ; 0x8000
48 b2: adds  r4, #1
49 strh.w  r3, [r5], #2
50 cmp r4, r7
51 bne.n 8e <cv::cvt32f16s(float const*, unsigned int, unsigned char const*, unsigned int
       , short*, unsigned int, cv::Size_<int>, double*)+0x8e>
```

Figure 3.6: Analysis of AUTO vs HAND vectorized ARM assembly code for Benchmark 2

of data to be fetched from main memory across all platforms unless the data is already resident in cache.

For the smallest image resolution of 640×480 with a file size of 1.2MB, several platforms are able to fit such an image in one of their cache levels. For example, the L2 cache of the Intel Core2 Quad, the L3 Cache of the Intel Core i5, the L2 cache of both ARM Cortex-A15 systems and the L2 cache of the ARM Cortex-A57 system are all able to fit a single 640×480 image. Not surprisingly, higher absolute performance is observed for the 640×480 images compared to other image sizes for these platforms across all the benchmarks presented in Table 3.2 and 3.3.

Normalizing base frequency across ARM A15 platforms, we observe that the TI Keystone II has higher absolute performance compared to the Jetson TK1. Given that we observe the Keystone II DDR3 read latencies to be approximately 100 cycles or 40% lower compared to the TK1, higher performance is expected from the Keystone II.

All ARM platforms other than the TI Keystone II A15 have an L2 cache size of 2 MB or less. The TI Hawking has a 4 MB L2 cache. This is another reason the Keystone II A15 has higher performance normalized w.r.t frequency compared to TK1 A15 for all the benchmarks.

Considering the impact, a memory controller's efficiency may have on SIMD performance, we compare the Exynos 4412 A9 to the Tegra T30 A9, both running at 1.4GHz. The combination of the Exynos 4412 interconnect and LPDDR2 RAM performs around 5× faster than the Tegra T30 with DDR3L for stride 16 byte read operations from main memory. From Table 3.3, the higher performance of the Exynos 4412 A9 can be attributed to the large differences in main memory read latencies observed in figures 3.3(a)-3.3(d).

3.4.3 Comparing generations of ARM processors

The ARM platforms evaluated in our experiments span across two architecture generations, ARMv7 and ARMv8, along with four processor generations, Cortex-A8, A9, A15 and A57. While the A8 systems are in-order, the other ARM processors are out-of-order. The A57 is the only 64-bit processor while all others are 32-bit.

One expects the performance of processors to improve for newer generations. This is indeed the case and a steady increase in performance going from older to newer ARM processor generations is observed. As expected the A57 performed better than the other ARM systems across all benchmarks.

Normalizing clock frequency and comparing Cortex-A9 cores to Cortex-A8

cores, we see HAND performance remains consistent while AUTO performance increases 1.1× on average across benchmarks. Comparing Cortex-A9 to Cortex-A15 cores, we see HAND performance increase 1.4× while AUTO performance increases 1.7× on average. Finally, Cortex-A57 compared to Cortex-A15 shows HAND performance dramatically increases 3.7× while AUTO increases by 1.3×.

3.5 Summary

The primary objective of this chapter was to experimentally establish whether the use of compiler intrinsic functions on ARM CPUs yield significantly higher performance compared to relying on compiler auto-vectorization. This was indeed shown using OpenCV application benchmarks of varying complexity across multiple ARM platforms. In addition, this was also found to be true for Intel platforms.

Analysis of the assembly code for the data-parallel benchmark 2 showed that auto-vectorization required more instructions per pixel than intrinsics, failing to treat multiple pixels at a time. For each benchmark as the gcc compiled hand-optimized intrinsic code and the auto-vectorized code was the same across all the Intel platforms and across all the ARMv7 platforms, similar AUTO:HAND speedup ratios might be expected within each group of processors (with possible differences between the Intel and the ARM processor groups). Our results showed that this was not always the case. Measurement of memory read latencies across selected Intel and ARM platforms provided some explanations for these cases.

Contributions and findings described in this chapter are as follows,

- Direct performance comparison between ARM NEON and Intel SSE2 intrinsic optimized code and their relative impact with respect to auto-vectorized code across nine ARM and four Intel platforms.
- Use of ARM NEON compiler intrinsic functions yielded between 1.05× and 13.88× faster code compared to compiler auto-vectorization. In comparison, use of Intel SSE intrinsic functions yielded between 1.34× and 5.54× faster code compared to compiler auto-vectorization.
- Effectiveness of using NEON intrinsic functions increased 1.4× for ARM Cortex-A15 cores compared to Cortex-A9 cores, while it increased 3.7× for Cortex-A57 cores compared Cortex-A15 cores.
- The latest generation of 64-bit ARM processors with ARMv8 architecture demonstrates higher performance compared to all other 32-bit ARM sys-

tems evaluated. However, the 64-bit ARM processor had lower performance across all benchmarks compared to the Intel Ivy-bridge processor.

As future work, extending this analysis to the other benchmarks and investigating whether other compilers are better able to auto-vectorize these codes is of interest. Extending the latency measurements for all evaluated platforms requires further work, especially for the Android smart-phones. Also, using hardware performance counters to measure instruction thorough-put and cache miss rates would be of interest.

Chapter 4

Development and Implementation of OpenMP 4.0 on the Keystone II SoC

Chapter 3 investigated the use of SIMD operations to maximize performance of ARM CPU cores on LPSoC systems including the Keystone II (K2). This chapter considers the other compute device on an LPSoC, the on-chip DSP accelerator, and considers the suitability of OpenMP as a programming model for the accelerator.

One of the main advantages of an LPSoC is the inclusion of an on-chip accelerator. Given a data or task parallel computation, such an accelerator can be put to optimal use alongside the host CPU by offloading independent parts of the computation to it.

Use of GPUs as separate attached accelerators has gained popularity over the last decade due to existence of efficient and easily available HPC programming models allowing such computation offload. The Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) are prime examples.

DSP accelerators have traditionally not been used for HPC workloads and therefore not gained traction in the HPC community. This perspective changed in 2012 when the TI Keystone I (K1) C66x multi-core DSP was shown to provide higher energy efficiency (GFLOPS/Watt) for SGEMM matrix multiplication compared to contemporary HPC systems [Igual et al., 2012].

With the introduction of the quad-core general purpose ARM CPU alongside DSP cores in the K2 SoC, a more fully featured HPC building block was formed. As a result, hardware vendors such as nCore HPC, Prodrive, and HPE integrated Keystone SoCs in commercially available products in early 2013.

nCore HPC, a silicon valley start-up company, in collaboration with a European conglomerate Prodrive Technologies, designed and manufactured the first rack-unit form-factor HPC blade containing the K2. This system known as the nCore *BrownDwarf* incorporated one K2 and two K1 SoCs connected via Hyper-

link in a single node. Meanwhile HPE, a global conglomerate, integrated four K2 SoCs in its Moonshot m800 compute server cartridge connected to each other via Hyperlink.

The Keystone architecture and consequently the nCore BrownDwarf and HPE Moonshot system faced significant impediments to their uptake. These included i) lack of performance evaluation of the Keystone DSP processor and comparison with conventional HPC systems; ii) lack of support for HPC programming models that allowed offloading computation from the host CPU cores to the DSP processors in the K2; and for the BrownDwarf system iii) lack of a software framework to offload computation from the K2 SoC to the two K1 SoCs.

The work presented in this chapter addresses the first two issues. Specifically, contributions of this work are as follows:

- Performance evaluation of an alpha version of TI's bare-metal OpenMP runtime for the C66x multi-core DSP
- First successful mapping of OpenMP 4.0 compiler directives to OpenCL runtime library operations
- Design, implementation and evaluation of the OpenMP 4.0 *accelerator model* runtime for K2 SoC to allow code running on *host* ARM cores to offload computation to *target* on-chip DSP cores

Section 4.1 details our performance evaluation of TI's first bare-metal OpenMP runtime. Section 4.2 describes the design, implementation and evaluation of an OpenMP 4.0 runtime environment for the K2 SoC. Section 4.3 concludes with a summary of key contributions.

4.1 Evaluating the Bare-Metal OpenMP runtime on TI C66x DSP

Market adoption of any technology requires extensive performance validation and comparisons with contemporary solutions. This was lacking for TI's new implementation of a *bare-metal* OpenMP runtime. The work presented in this section addresses this issue by benchmarking the first alpha version of the bare-metal runtime and comparing it against other systems.

The OpenMP runtime used in [Igual et al., 2012] was built above the SYS/BIOS RTOS which had significant overheads. To increase performance, TI implemented a lightweight OpenMP runtime using the Open Event Machine RTOS

PROCESSOR	CODENAME	Threads/Cores/Ghz	Memory
Intel Core2 Q9400	YorkField	4/4/2.67	8GB DDR3
Intel Xeon X5650	Westmere	12/6/2.67	24GB DDR3
Samsung Exynos 4412 (ARM)	Odroid-X2	4/4.Cortex-A9/2.0	2GB LPDDR2
TI Keystone II (ARM)	Hawking	4/4.Cortex-A15/0.625	2GB DDR3
TI Keystone II (DSP)	Hawking	8/8.C6678 DSP/0.983	2GB DDR3

Table 4.1: Platforms used in benchmarks to evaluate bare-metal OpenMP runtime

which provided direct access to the hardware queues for inter-process communication (IPC) between the DSP cores. Since IPC was now provided across atomic hardware queues, this runtime promised significant performance improvements compared to the previous SYS/BIOS implementation which used spin locks implemented on shared memory.

The EPCC v3 benchmark [Bull et al., 2012] was used to evaluate the performance of TI's OpenMP runtime. Comparison data was collected across other contemporary ARM and Intel platforms. The EPCC suite of micro-benchmarks measure the time overheads associated with invoking the different OpenMP constructs, for example, the cost of *parallel* to create a parallel region or *barrier* to synchronize threads. Here we report the overheads associated with some of the most widely used constructs.

4.1.1 Hardware Platforms

Table 4.1 lists the systems considered and their main characteristics. Two different Intel platforms with at least four physical cores were considered. The Core2 Q9400 Yorkfield processor, has four cores running at 2.66 Ghz. The hexa-core Xeon X5650 Westmere processor is part of a dual-socket system and runs at 2.66 Ghz. The ARM platforms considered, include the K2 EVM's ARM Cortex A15 quad-core processor running at 625 Mhz and a Samsung Exynos 4412 prime SoC with quad-core ARM Cortex-A9 processors running at 2 Ghz. The ARM Cortex A15 is referred to as K2-A15 and the A9 as Exynos-A9. The octa-core C66X DSP processor in K2 EVM ran at 983 Mhz and is referred to as K2-DSP.

4.1.2 Compilers and Tools

For the Intel Westmere platform we used GCC 4.6.4 and ICC 13.1.1 (compatible with GCC 4.6) to compile separately and run the benchmarks. These versions are denoted as X5650-GCC and X5650-ICC. They were linked against libgomp and libiomp5 respectively. On the Intel Yorkfield and ARM platforms GCC 4.7.3

with libgomp was used. The compiler option `-mcpu=cortex-a9` was used on the Exynos and `-mcpu=cortex-a15` on the K2. In addition, both ARM platforms used the `-mfpu=neon,mfloat-abi=hard` compiler flag.

TI Code Generation Tools 7.4.2, XDC Tools 3.24.05.48, OpenEM 1.2.0.1, PDK Keystone2 1.00.00.09, PDK C6678 1.1.2.6 along with the alpha version of the OpenMP runtime were used to create the executable for the C6678 DSP.

All platforms except the K2-A15 and the DSPs were running Ubuntu Linux with kernel version greater than 3.0. The K2 ARM cores used a custom distribution of Linux, called Arago, built specifically for the K2 EVM. It includes the 3.8.4 Linux kernel. On the Linux hosts, the `OMP_PROC_BIND` and `GOMP_CPU_AFFINITY` environment variables were set to bind threads to processor cores and to prevent thread migration between cores.

For timing measurements on the Intel and ARM platforms, the EPCC v3 timer function `getclock()` which uses `omp_get_wtime()` remained unchanged and had microsecond resolution. Measurements on the DSP required modifications to the timer function. A native time-stamp counter was used to measure the exact CPU cycles elapsed as shown in Figure 4.1. For direct comparison of all platforms, all time measurements were normalized w.r.t CPU clock speed and reported in CPU cycles using the equation, cpu_cycles $=$ overhead_time$(\mu s) *$ mhz.

4.1.3 Results

A crucial difference between the multi-core DSP in the Keystone architectures and other processors evaluated in this study is cache coherence. While the Intel and ARM multi-core processors have hardware managed cache coherence protocols, programs running on the multi-core DSP have to ensure cache coherence in software.

The Keystone shared memory controller does not ensure memory consistency across DSP cores. As a result, the runtime performs *flush* operations at implicit and explicit synchronization points which invalidate L1 and L2 caches and write them back to main memory. In our evaluation, DSP L2 cache was set to be 0K, i.e. L2 memory was set to entirely act as scratchpad RAM (SRAM). This was performed using the TI Chip Support Library (CSL) API function, `CACHE_setL2Size(CACHE_0KCACHE)` to minimize flush overhead. Table 4.2 presents operation cycle counts on the K2 DSP averaged over 200 iterations. This shows the cost to be roughly 1350 cycles regardless of thread count.

Figure 4.2 presents results of the EPCC benchmarks. In each bar-graph the computational overhead is measured in CPU cycles and presented for each plat-

```
/* Wall cycles using TSC_read */
void wcycles(unsigned long long *c)
{
    static int first = 1;
    extern void TSC_enable(void);
    extern unsigned long long TSC_read(void);
    if (first)
    {
        TSC_enable();
        first = 0;
    }
    *c = TSC_read();
}

/* TSC_enable Assembly Code */
.global TSC_enable

TSC_enable:

RETNOP  B3, 4
MVC     B4, TSCL  ; writing any value enables timer

/* TSC_read Assembly Code*/
.global TSC_read

TSC_read:

RETNOP  B3, 2
DINT
MVC TSCH,   B5    ; Read the snapshot of the high half
MV  B5,  A5
```

Figure 4.1: Measuring CPU cycles on the DSP

DSP (L2 = 0)	1 Thread	2 Threads	4 Threads	6 Threads	8 Threads
Keystone II DSP	1350	1355	1357	1353	1364

Table 4.2: Cost of software managed cache coherency operation for DSP (cycles)

form using 1-8 OpenMP threads. For platforms with 4 cores, only results with up to 4 threads are given.

Consider first Figure 4.2(a) which presents overheads for the OpenMP PARALLEL construct. This is the most fundamental construct which specifies the creation of an OpenMP parallel region and spawning of a team of threads. Each of the platforms demonstrates the expected behavior of increasing overhead of CPU cycles with increasing number of threads.

For the single thread performance, and the Intel platform, the X5650-ICC reports the least overhead of 547 cycles which is marginally lower than the other two Intel systems. Among the ARM based platforms, the K2-DSP reports the least overhead of 779 cycles which is less than half the overhead of the other two ARM platforms. The overheads measured across all the Intel systems are, however, marginally less than the K2-DSP and 3× to 4× less than the other ARM platforms.

For the multi-thread performance across the Intel platforms, overheads increase near linearly except the 8-thread overheads for the X5650 measurements. This is expected because the X5650 has 6 physical cores. Up until the 6-thread measurements, the overheads increase linearly since these threads are each executed separately on a physical core. The two extra threads in the 8-thread measurement are executed on hyperthreads multiplexed on two physical cores with two other hyperthreads.

Among the ARM platforms, the Exynos-A9 and K2-A15 show near linear increase in overheads from 1-thread to 2-thread measurements. However, their 2-thread and 4-thread overheads are almost equal. This suggests that the gcc OpenMP runtime on ARM systems might have equivalent thread creation overhead when creating more than two threads. On the K2-DSP, overheads observed for multiple thread creation are at least 7.5× higher compared to 1-thread. This is because each new thread creation incorporates implicit cache-coherence flush operations.

Comparing across Intel and ARM measurements, Exynos and K2-A15 ARM processors report comparable overheads to the Intel platforms. The K2-DSP bare-metal runtime, however, reports 1.5× to 4.5× higher multi-threading overhead compared to all other platforms. If the cost of software managed cache-flush operations on the K2-DSP (table 4.2) were removed, the overheads would be comparable.

The overhead of the OpenMP BARRIER construct is shown in Figure 4.2(b). This construct is used to specify an explicit synchronization point inside a parallel region which all threads must reach for any of them to progress beyond that point. The X5650-ICC performs the best among all platforms across all thread

configurations. Similar to PARALLEL results, the ARM platforms have comparable overheads to Intel. The K2-DSP has overheads of between 1800 and 3206 cycles. Subtracting the cost of 1 flush operation from these yields overheads of between 450 and 1842. The latter are slightly larger than the X5650-GCC values.

Figure 4.2(c) presents overheads of the FOR construct. This construct is used to split iterations in for loops between OpenMP threads. Results in this figure show very similar patterns across all platforms to those observed for both the PARALLEL and BARRIER benchmarks.

Figure 4.2(d) and 4.2(e) present overheads for two different STATIC directives. These are used to specify compile-time scheduling of loop iterations between threads. Specification of the *static* construct differs from the default scheduling of the *for* construct in that it specifies a block size for the schedule. If not specified, the scheduled block size is dependent on the runtime implementation. STATIC 1 indicates that each thread gets 1 loop iteration to process at a time, while STATIC 128 gives 128 loop iterations at a time.

Results from the STATIC benchmarks show that K2-DSP performs significantly better than the Intel platforms, while the ARM platforms perform the best overall. This suggests that the chunk sizes of 1 and 128 are not ideal or too small for the memory hierarchy of the Intel platforms, but are more suited for the ARM and DSP platforms for this particular benchmark. Results also show that the K2-DSP multi-thread overheads for STATIC 128 are significantly lower compared to STATIC 1. This indicates that higher block sizes which lead to lower number of scheduling blocks reduces scheduling overhead on the K2-DSP.

The DYNAMIC construct is similar to STATIC in that it partitions the scheduling of loop iterations between threads. In contrast to STATIC the loop iterations are now partitioned dynamically with the next available thread executing the next loop iteration. This permits load balancing when iterations give rise to variable work. The DYNAMIC 1 benchmark therefore quantifies the overhead of pulling work items off a shared global queue at runtime compared to a predefined set of work items at compile-time.

DYNAMIC 1 overheads are reported in Figure 4.2(f). Across the Intel platforms, DYNAMIC 1 overheads are very similar to STATIC 1 indicating that shared queue access overhead was negligible. Across the ARM platforms, shared memory is used for storing global queues. When shared memory is used, the locking involved for DYNAMIC 1 would incur some overhead compared to STATIC 1. The results for ARM systems confirm this expectation.

On the K2-DSP, the bare-metal OpenMP runtime uses hardware queues to implement shared global queues. It is expected that DYNAMIC 1 overhead would be constant across all threads since each atomic transaction on a hard-

ware queue is deterministic and takes constant time. This expectation is again confirmed in Figure 4.2(f). Note the 1-thread overhead is also similar to the multi-thread overhead for DYNAMIC 1. In comparison, the 1-thread overhead is an order of magnitude lower for STATIC 1. This is consistent with the use of a global queue for DYNAMIC 1 from which work items are pulled even when a single thread is processing all items.

4.1.4 Analysis

The overheads measured for PARALLEL, BARRIER and FOR constructs show that one of the primary limitations of the Keystone architecture is the absence of hardware cache coherence on DSP cores. The implementation of the bare-metal OpenMP runtime performs necessary cache operations at flush points as required to maintain consistency and correctness of results. A significant computational cost of approximately 1350 CPU cycles must be paid when each OpenMP synchronization or flush point is encountered.

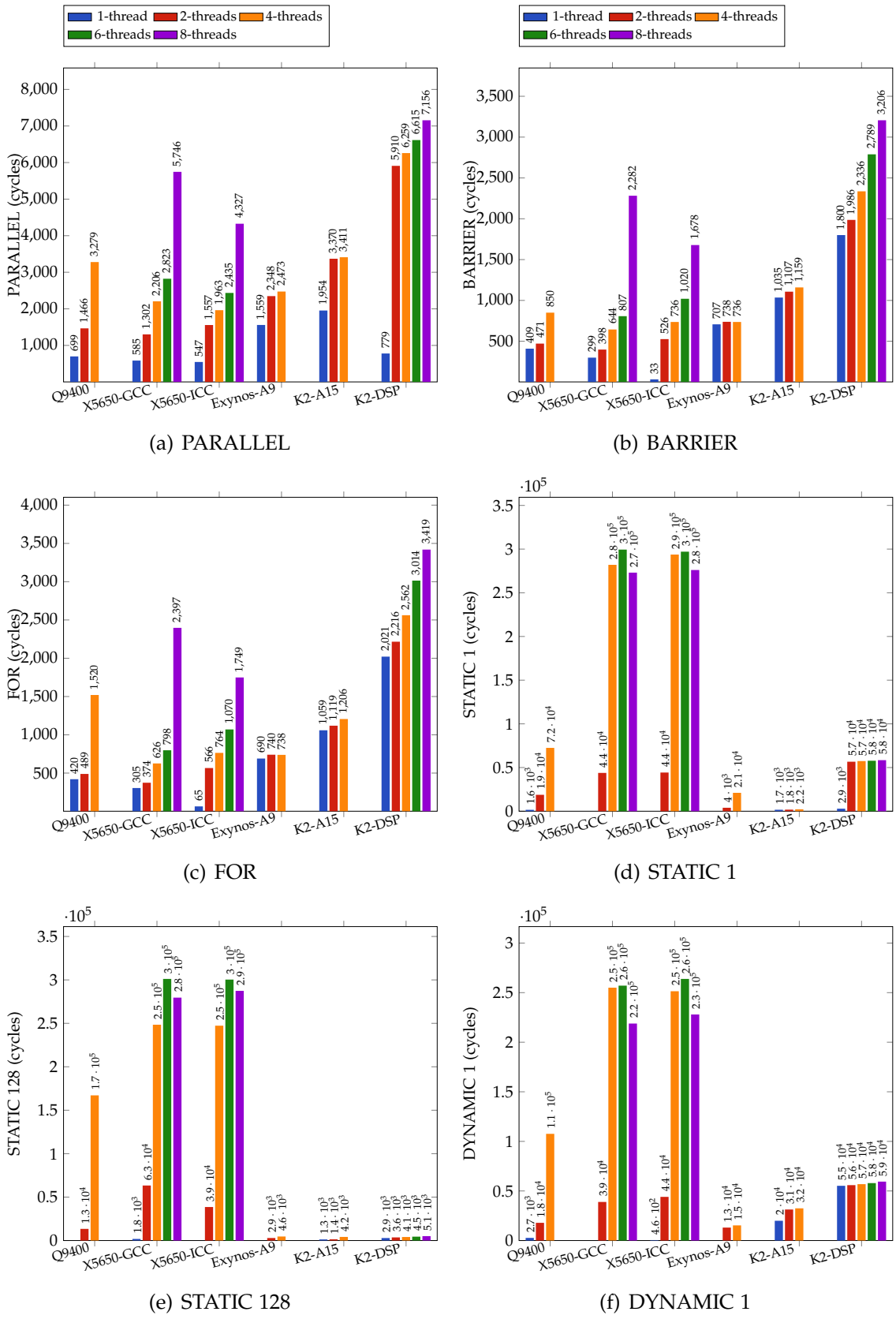
The EPCC benchmark results demonstrate that the bare-metal K2-DSP runtime performed at par or better with Intel and GCC OpenMP runtimes across OpenMP constructs benchmarked when the cost of software managed cache coherence is considered. The obvious implication is to try and minimize the number of synchronization points in program logic to avoid the cache coherence overhead on the K2-DSP. Similarly, from the STATIC and DYNAMIC overheads, one should try and schedule loops with larger block sizes for higher performance on the K2-DSP.

This bare-metal runtime paves the way for applications with OpenMP code sections to be implemented on the K2 C66x DSP cores. However, coordinating the use of these DSP cores from host K2-ARM cores required implementation of a new runtime to provide OpenMP 4.0 constructs as described in the next section.

4.2 OpenMP 4.0 on Keystone II

One of the main advantages of an LPSoC such as the K2 SoC is shared physical memory which makes it easier for both CPU and accelerator cores to work simultaneously on a single problem. The main objective of work presented in this section was to exploit this vital feature of an LPSoC using OpenMP 4.0 to offload computation from the ARM CPU to the DSP cores on the K2 SoC

The OpenMP 4.0 *accelerator model* specification, released in 2013, permits a workflow of offloading computation from a *host* to a *target* processor through



(a) PARALLEL (b) BARRIER

(c) FOR (d) STATIC 1

(e) STATIC 128 (f) DYNAMIC 1

Figure 4.2: Cost comparison of OpenMP constructs in CPU cycles

the use of a compiler pragma. This simplifies programming of an application greatly as a single OpenMP pragma can be used both to offload a code segment and to orchestrate data transfers to and from an accelerator. Features of the OpenMP programming model and version 4.0 in particular are discussed in § 2.4.1. An efficient implementation of OpenMP 4.0 on the K2 adds significant weight to the utility and ease of use of the heterogeneous K2 LPSoC and its expected performance efficiency.

An implementation of an OpenMP 4.0 runtime for the K2 SoC could be developed from scratch using low-level drivers for accessing hardware queues and shared memory between ARM and DSP cores along with drivers to boot, start and stop DSP cores from ARM cores. This manner of implementation would ensure high efficiency since K2 platform low-level drivers would be used directly. However, this would make the implementation non-portable and specific to the K2 SoC.

Another possibility was to build the OpenMP 4.0 runtime above TI's OpenCL [Khronos, 2011] runtime by using library API. A disadvantage of using the OpenCL runtime would be the overheads incurred compared to directly using low-level drivers. Another disadvantage would be that OpenCL versions before v2.1 restrict kernels to C code. Therefore, an OpenCL library version less than 2.1 would restrict the specification of offloaded OpenMP code to C code only.

The primary advantage of using OpenCL would be cross-platform portability of this OpenMP 4.0 runtime. Since it would be using a standardized OpenCL API, any LPSoC platform with an OpenCL conformant library could be a potential target of this OpenMP runtime. The definition of a functional mapping between OpenMP 4.0 constructs and OpenCL library operations could then be re-used on any heterogeneous system. Given this advantage, a decision was made to use OpenCL as a back-end rather than directly using low-level drivers. The possibility of incurring runtime overheads associated with using OpenCL and being restricted to C kernels were outweighed by cross-platform portability and wide applicability of the OpenMP to OpenCL mapping.

TI provides an OpenCL version 1.1 conformant runtime library for the K2. This OpenCL library can be used on the ARM cores of the K2 to create *buffers* in shared physical memory, populate these buffers with input data, execute OpenCL kernels on DSP cores operating on these buffers and finally read results back on the ARM cores. Mapping OpenMP 4.0 to OpenCL has the potential to encapsulate multiple lines of boilerplate OpenCL library calls in compiler directives while making the data transfers seamless.

Leveraging other unique hardware features available in the K2 SoC for improved performance was another objective of the work presented in this section.

Specifically, how to use the local DSP L2 SRAM memory through OpenMP 4.0 constructs and how to leverage DMA co-processors for fast memory transfer were investigated. A new OpenMP data map-type *local* was proposed and implemented to achieve this.

To evaluate the performance of the OpenMP 4.0 runtime the matrix multiplication (GEMM) kernel originally written for the K1 DSP cores [Igual et al., 2012] was programmed to use the DSP cores on the K2 SoC using our implementation of the OpenMP 4.0 runtime alongside other performance optimizations.

Section 4.2.1 describes the OpenMP 4.0 to OpenCL mapping defined and factors considered for it. Section 4.2.2 describes the design and implementation of the OpenMP 4.0 runtime environment for the K2 SoC. Issues and challenges encountered in the implementation of GEMM using our OpenMP 4.0 runtime are discussed along with its performance in § 4.2.5. The importance of leveraging unique K2 hardware features such as using buffers in local SRAM, transferring data using DMA co-processors, allocating buffers in memory shared between the ARM and DSP cores are highlighted in § 4.2.3 and § 4.2.5. Trade-offs and factors affecting performance are also discussed.

4.2.1 Mapping OpenMP 4.0 to OpenCL

Both OpenMP 4.0 and OpenCL define a system architecture composed of a single *host* CPU device offloading computation to one or more *target* devices. The memory model definition, however, is somewhat different. OpenCL is designed to operate on a distributed address space. OpenMP 3.0 is designed to operate on a shared address space, while OpenMP 4.0 assumes a distributed one. This is an advantage for LPSoC systems since most such systems do not have host and target devices sharing address spaces due the lack of a common MMU. The K2 is no exception and therefore both OpenCL and OpenMP 4.0 memory models apply directly.

Even though the OpenMP 4.0 memory model, described in § 2.4.1, is designed for a distributed address space, shared physical memory on LPSoC systems such as the K2 enable the *map* clause to signify cache operations rather than data transfers. It is trivial to reserve shared memory at an operating system level to be usable by both ARM and DSP cores on K2. This allows data transfer operations to be reduced to simple cache coherence operations. For example, when ARM cores write to a shared buffer, only an ARM cache flush is required before the data is ready for DSP cores to use. TI also provides a contiguous memory allocator, CMEM (see § 4.2.3), which is able to allocate memory in such reserved memory regions and handle necessary cache operations.

Four new directives introduced in the OpenMP 4.0 *accelerator model* specification are supported in this runtime implementation. These are:

- `#pragma omp target`
- `#pragma omp declare target`
- `#pragma omp target data`
- `#pragma omp target update`

The *teams* and *distribute* directives are not supported in this implementation. These constructs were primarily introduced in OpenMP to cater for GPU accelerators which launch thousands of lightweight threads. Since only eight threads are launched on eight DSP cores available on K2, the objective of this runtime implementation was simply to cater for a single team of threads running across all eight DSP cores each time an OpenMP target region is executed.

The *target* clause signifies the intention to use a *target* device i.e. an accelerator device. Using the `#pragma omp target` directive, a region of code can be offloaded to an accelerator device from the host CPU. The other three directives can be used to setup and manage data environments on the accelerator device for use during the execution of a target region. Section 2.4.1 describes these directives in more detail.

The purpose of having a concrete mapping between OpenMP 4.0 compiler directives and OpenCL library API calls is cross-platform portability of the mapped functions. For example, if an OpenMP 4.0 compiler directive is translated by a compiler on platform A to a set of mapped functions, this translated code should be possible to link with mapped functions implemented on platform B and execute on platform B.

For this reason, we define a set of functions that OpenMP 4.0 constructs map to as shown in Figure 4.3. The GCC compiler work offload API [GNU, 2017] naming convention is used to define these functions. The objectives of each of these mapped functions are as follows:

- `GOMP_target`: Create a device data environment and trigger execution on target device. Execute when *target* directive is encountered with data objects and transfer directions specified in *map* clause. Device data environment is removed at the end of this call and all data objects are de-allocated.
- `GOMP_target_data`: Creates a device data environment using data objects specified in the *map* clause. This data environment persists until explicitly destroyed.

- `GOMP_target_data_end`: Destroys an existing device data environment and de-allocates its data objects.
- `GOMP_target_update`: Make any data environment on the target device consistent with the host i.e. makes data computed within a target region available to the host

```
/****************************************************************/
/* Create a target region on the given device and execute the given*/
/* function on it                                               */
/****************************************************************/
void GOMP_target (int device,
                  void (*fn) (void *),
                  char          *fnname,
                  unsigned int   mapnum,
                  void         **hostaddrs,
                  unsigned int  *sizes,
                  unsigned char *kinds);

/****************************************************************/
/* Setup data items specific to a target data region            */
/****************************************************************/
void GOMP_target_data (int            device,
                       unsigned int   mapnum,
                       void         **hostaddrs,
                       unsigned int  *sizes,
                       unsigned char *kinds);

/****************************************************************/
/* Destroy data for a target data region                        */
/****************************************************************/
void GOMP_target_data_end (int            device,
                           unsigned int   mapnum,
                           void         **hostaddrs,
                           unsigned int  *sizes,
                           unsigned char *kinds);

/****************************************************************/
/* Make a target update for given data items                    */
/****************************************************************/
void GOMP_target_update   (int            device,
                           unsigned int   mapnum,
                           void         **hostaddrs,
                           unsigned int  *sizes,
                           unsigned char *kinds);
```

Figure 4.3: OpenMP 4.0 to OpenCL: mapped functions

While these functions represent another layer of abstraction above OpenCL, they can just as easily be implemented using low-level drivers. For the K2 SoC, these functions are implemented with OpenCL and constitute the ARM host library implementation of OpenMP 4.0, and will be described in § 4.2.2.2.

4.2.2 OpenMP 4.0 runtime environment

The fundamental requirements of an OpenMP 4.0 runtime implementation for an accelerator are, i) use of host CPU operations to orchestrate data movements to and from target accelerator device; ii) a runtime environment on the target device able to execute OpenMP C kernels and; iii) a bi-directional communications channel between the host CPU and target accelerator. The existing OpenCL v1.1 runtime for K2 provides operations satisfying all three of these requirements.

The OpenMP 4.0 runtime environment constitutes a compiler, runtime library and a meta-compiler. The objective of the compiler is to translate or *lower* source code containing OpenMP directives to mapped functions defined in § 4.2.1. More specifically, this compiler is known as a *source-to-source* translator. The objective of the runtime library is to provide an implementation of the mapped functions using OpenCL. The objective of the meta-compiler is to integrate both the compiler and runtime library aspects of the environment to produce an executable binary from input source code. All three of these components are intended to be separate to each other with different code-bases. The purpose of this is to maintain loose-coupling for portability so that re-using individual components is easily possible across different platforms.

Figure 4.4 describes these three components of the runtime environment. Details of each component are as follows:

- **omps2s**: Use of the *cl6x* front-end source-to-source translator to lower OpenMP 4.0 source code to library calls against *libOpenMPAcc* and generates an OpenCL kernel
- **libOpenMPAcc**: A host library that provides a thin layer above OpenCL
- **clacc**: A meta-compiler that takes OpenMP 4.0 source code, uses omps2s to lower target regions, compiles DSP code, ARM code separately and generates a self-extracting *fat* binary

4.2.2.1 Source-to-source translator: omps2s

To perform source-to-source lowering, a front-end of TI's proprietary C/C++ compiler *cl6x*, known as omps2s was used. Its purpose is to translate all OpenMP

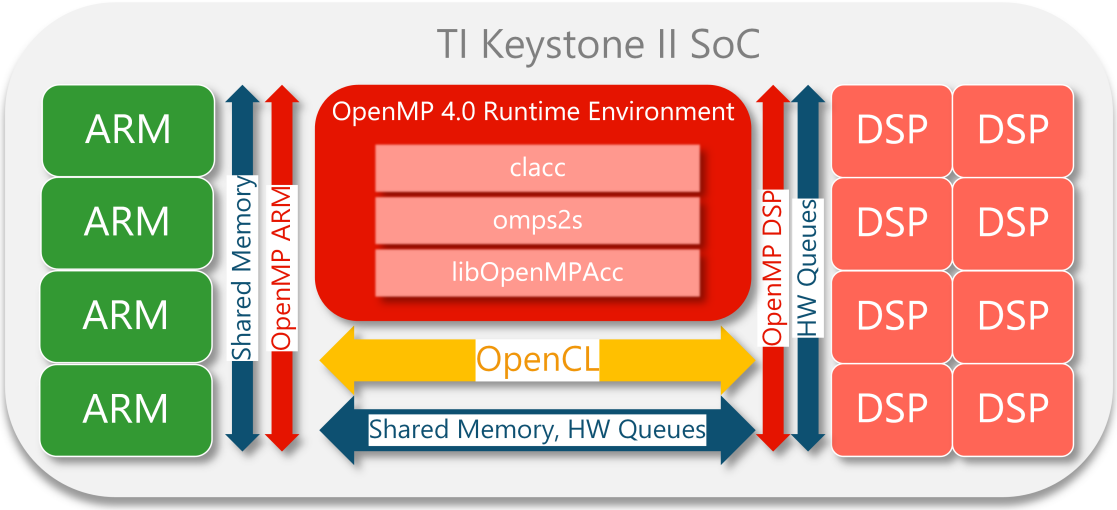


Figure 4.4: OpenMP 4.0 environment components

constructs including older OpenMP 3.0 constructs alongside 4.0 constructs. Essentially this type of front-end is standard in a platform's compiler suite such as gcc's OpenMP front-end triggered by –fopenmp. Omps2s takes as input a source file with OpenMP 4.0 constructs and generates three artifacts:

- Lowered host CPU code with calls to the mapped functions as defined in Figure 4.3. The host library, libOpenMPAcc, services these calls.
- Lowered target DSP code with calls to the standard OpenMP runtime API such as `GOMP_parallel_start` and `GOMP_parallel_end`. These calls are serviced by the TI bare-metal OpenMP runtime for the DSP cores.
- OpenCL kernel wrapper that encapsulates the lowered target DSP code along with placeholders to setup required data buffers

This process is illustrated in Figure 4.5. The source function `vadd_openmp` with an OpenMP target region is first translated to the lowered host function with the `GOMP_target` API call at the end. It is then lowered to the `__omp_1_vadd_openmp` target DSP function containing calls corresponding to the `#pragma omp parallel` directive i.e. `GOMP_parallel_start` and `GOMP_parallel_end`. Finally, the OpenCL wrapper `__kernel void __omp_1_vadd_openmp_wrapper` is generated which calls the target DSP lowered function.

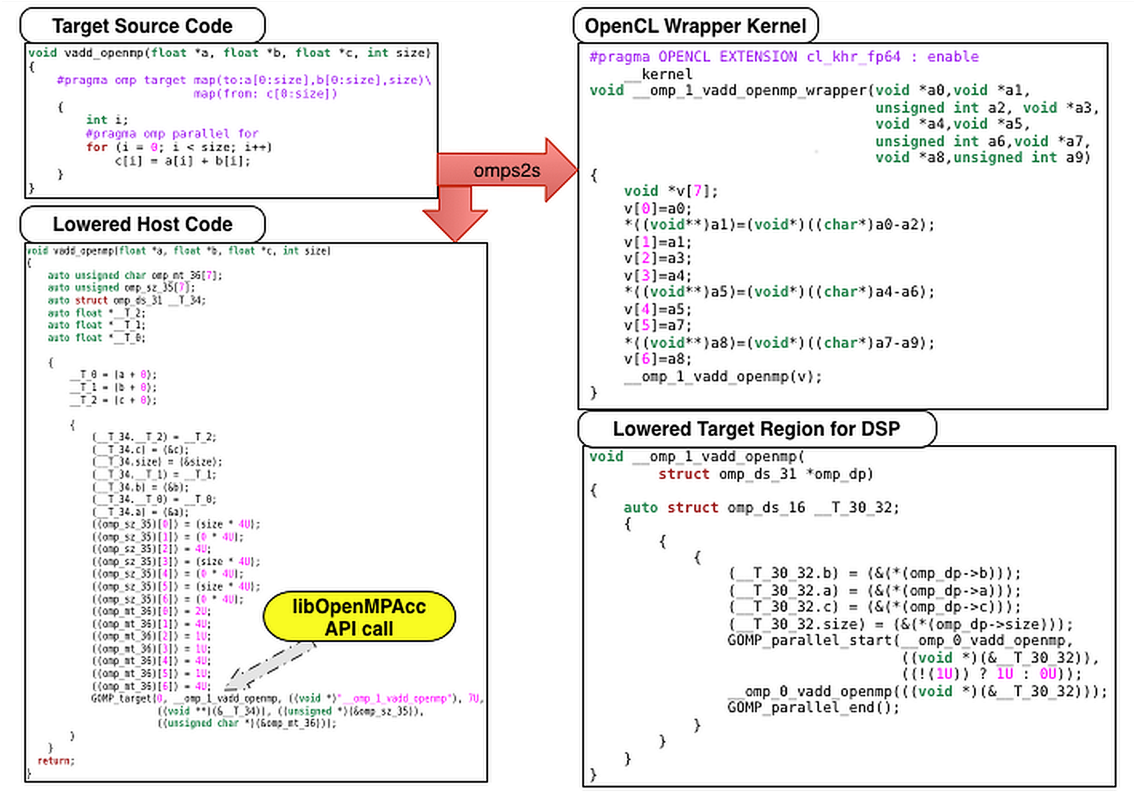


Figure 4.5: OpenMP 4.0 source-to-source lowering using omps2s

4.2.2.2 Host library: libOpenMPAcc

The purpose of this library is to provide the implementations of the mapped functions as defined in § 4.2.1. *libOpenMPAcc* is implemented in C++ as a thin layer above OpenCL effectively providing the low-level OpenCL API mappings to the OpenMP 4.0 constructs.

TI's OpenCL 1.1 compliant runtime implements an extension which supports dispatching OpenCL kernels with calls to functions containing OpenMP parallel regions. This feature allowed leveraging the OpenCL runtime to perform the mechanics of making data available on the target device and triggering execution of OpenMP regions.

The namespace *oa* signifying OpenMP Accelerator was used to encapsulate all classes contained in the library. The high-level design of libOpenMPAcc is given in Figure 4.6 and its corresponding low-level detailed design is provided in Figure 4.7. There are four distinct layers in the object-oriented class design of libOpenMPAcc:

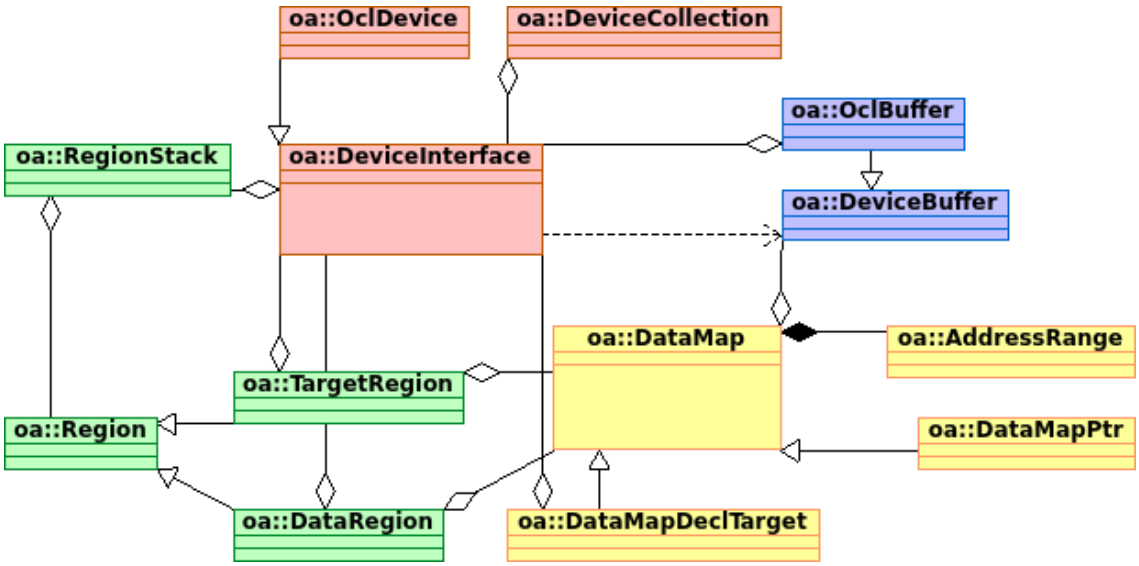


Figure 4.6: High-level Design: libompacc

- *Device Layer*: The `DeviceInterface`, `OclDevice` and `DeviceCollection` classes. They collectively provide instantiation, bootstrapping and setup and code compilation functions for target accelerator devices

- *Physical Memory Layer*: The `DeviceBuffer` and `OclBuffer` classes. They provide allocation of physical memory buffers on target accelerator device memory

- *Logical Memory Layer*: The `DataMap`, `DataMapPtr`, `AddressRange` and `DataMapDeclTarget` classes. They provide mapping of OpenMP data sections onto memory ranges inside physical memory buffers in target accelerator device memory

- *Execution Layer*: The `TargetRegion`, `DataRegion`, `Region` and `RegionStack` classes. They provide methods to instantiate and execute target and data regions on accelerator devices

The highest layer in the design is the execution layer which supports creating target and data regions corresponding to `#pragma omp target` and `#pragma omp target data` respectively.

Upon creation of a target region, a `GOMP_target` API call is invoked. A *TargetRegion* object is instantiated using the device id and supplied function parameters to be executed on the device. *DataMap* objects are then created to support

the OpenMP data buffers specified in the *map* clause using the `initializeMaps()` function.

The OpenCL device is then initialized using an *OclDevice* object instantiation via the `prepareExecutionObject()` function. Target DSP code and the OpenCL wrapper kernel generated by the source-to-source lowering stage is then just-in-time (JIT) compiled for the device. Data is then transferred to DSP memory using the `syncDataToTarget()` function.

At this point the target region is ready to be executed and this is triggered using the `TriggerExecution()` function which returns once the execution completes. Finally, result data is brought back to host memory using `syncDataFromTarget()`.

Upon creation of a data region, a `GOMP_target_data` API call is invoked. A *DataRegion* object is instantiated on the device specified. Similar to when a TargetRegion is created, *DataMap* objects are created to support *map* clause data items using the `initializeMaps()` function. These are then shipped to DSP memory using the `syncDataToTarget()` function.

The DataRegion is then *pushed* onto a *RegionStack* object since multiple DataRegions can be nested together. Only when the DataRegion's scope ends and a `GOMP_target_data_end` call is invoked; the data is synchronized with host memory using `syncDataFromTarget()` and then the corresponding DataRegion is *popped* from the RegionStack.

The `#pragma omp target update` directive invokes the `GOMP_target_update` call. This call creates another *DataRegion* object and uses the `updateMaps()` function to synchronize the host and device buffers as specified in the *map* clause.

4.2.2.3 Meta-compiler: clacc

The purpose of the meta-compiler, clacc, was to be a black-box tool that converted OpenMP 4.0 source code to an executable binary that a user could simply run on the host to execute the program.

The alternative was to expect a user to perform the compilation using omps2s and link against libOpenmpAcc and other required libraries manually to create two separate binaries, one for ARM and the other for DSP. These two binaries would then have to be executed in the correct sequence across both ARM and DSP cores for them to work correctly.

Clacc was designed to automate this entire process and create a single self-extracting *fat* binary with the user being entirely oblivious of the intermediate stages. When executed, the fat binary would then self-extract two separate ARM

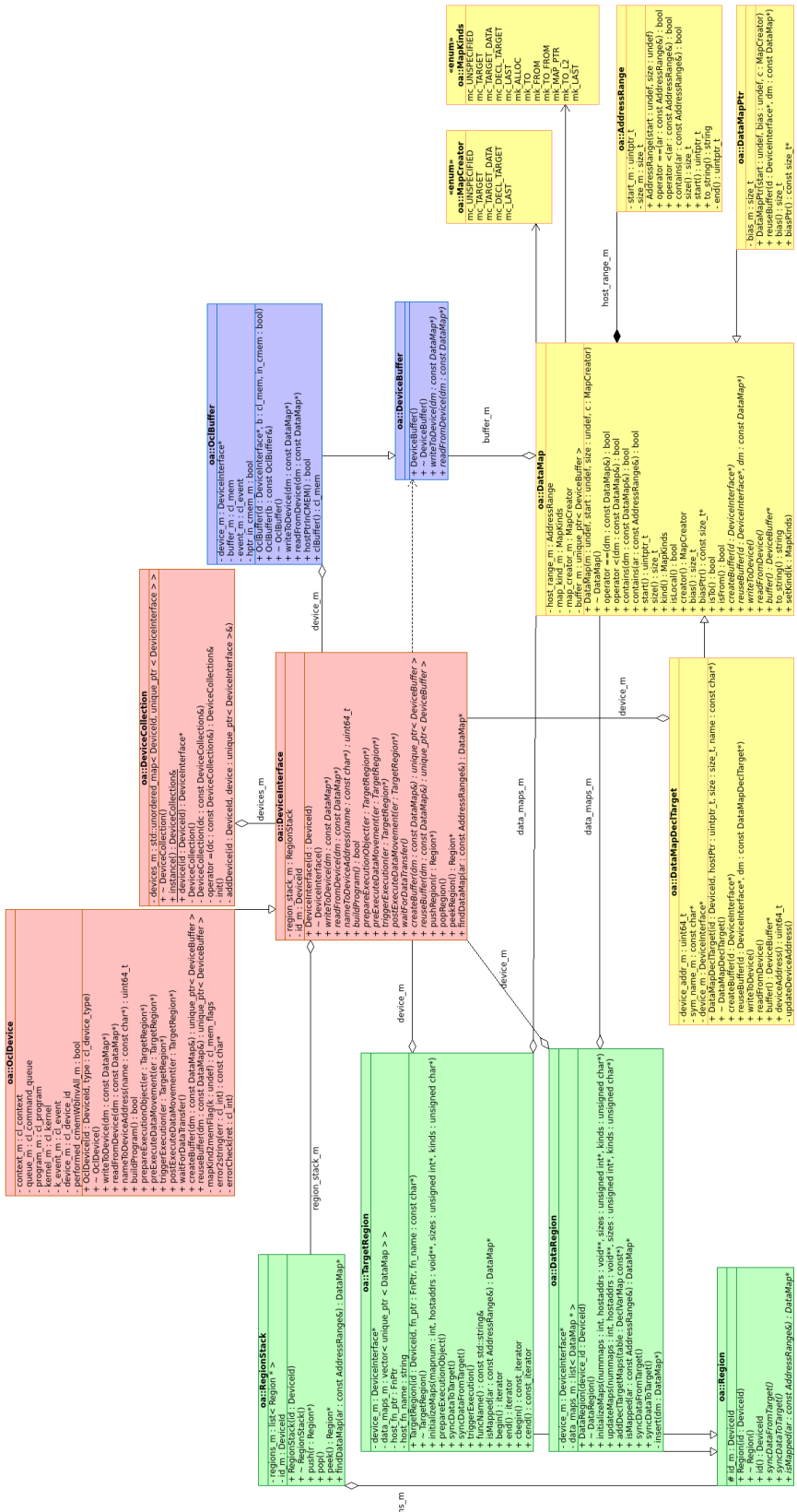


Figure 4.7: Low-level Design: libompacc

and DSP binaries and begin execution across both devices in the correct order.

Figure 4.8 describes the various stages involved in the compilation process managed by clacc. The right branch in the figure represents compilation of DSP code while the left side represents compilation of ARM code. Given OpenMP 4.0 target regions, an initial source-to-source lowering is performed for both ARM and DSP codes using the omps2s tool. This generates artifacts as described in § 4.2.2.1.

Consider the DSP code compilation. The TI *cl6x* compiler is invoked by clacc to generate the final DSP binary given omps2s DSP artifacts along with other dependent DSP objects and archive files. This binary is generated as an array embedded in a C file which is passed to the ARM side compilation.

On the ARM side, the *gcc* compiler is then invoked using the DSP device binary embedded C file, omps2s ARM artifacts along with the libOpenMPAcc host library. This generates ARM object files which now have in them an embedded DSP binary and host side calls serviced by libOpenMPAcc. These object files are passed to the gcc linker along with other dependent object files and/or archives. Finally, this stage produces a fat binary which contains both ARM and DSP binaries with glue code to bootstrap and launch each of them separately.

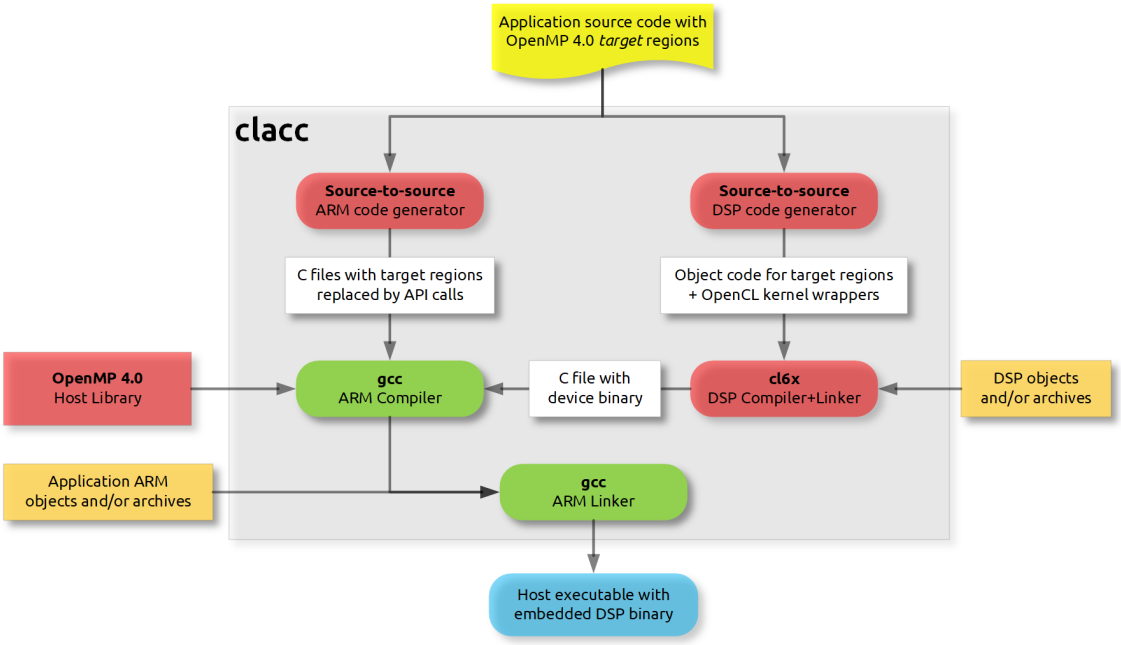


Figure 4.8: The CLACC meta-compiler

4.2.3 OpenMP Runtime Optimization: Allocating buffers in shared memory

Data synchronization between the host and target device can be a significant source of overhead. This overhead has implications for the amount of computation that needs to be performed by a target region to outweigh the data synchronization overhead. On the K2 SoC, the host and target device share both DDR RAM and MSMC SRAM. However as noted earlier: (a) the target device does not have a memory management unit (MMU); and (b) there is no hardware cache coherency between the target and host device.

As a result, the host accesses shared memory using virtual addresses and the target accesses the shared memory using physical addresses. Moreover, host device variables can span multiple non-contiguous pages in Linux virtual memory whereas the target device operates on contiguous physical memory. When mapping buffers from the Linux process space, the buffers must be copied into contiguous memory for target operation. This copy is inefficient, especially for large buffers. To eliminate this copy, a special purpose dynamic memory allocation API, `__malloc_ddr()` and `__malloc_msmc()` was implemented as part of libOpenMPAcc. Its usage is shown in Figure 4.9.

The physical memory associated with this heap is contiguous and is mapped to a contiguous chunk of virtual memory on the host. If any host variables allocated via this API are mapped into target regions, the map clauses translate to cache management operations on the host, significantly reducing the overhead.

```
1 float* buf_in_ddr  = (float*) __malloc_ddr(size_bytes);
2 float* buf_in_msmc = (float*) __malloc_msmc(size_bytes);
```

Figure 4.9: __malloc_ddr and __malloc_msmc API

In Figure 4.10 we report the advantage of mapping buffers allocated using the dynamic allocation API compared to those allocated using standard `malloc()` on the host. The overhead times of offloading buffers with sizes varying between 4 KB to 163840 KB (160 MB) created using both the `__malloc_ddr()` API and standard `malloc()` were measured. The speed-up obtained by using `__malloc_ddr()` is shown. A buffer was first read and written to in order to populate the host caches. Following this the host timer was started and the buffer was offloaded to a target region using `map(tofrom:buffer[0:size])`. Within the target region, all eight DSP cores wrote to the buffer using an OpenMP parallel region. Upon returning from the target region, timing was stopped and

recorded on the host. The elapsed DSP time measured within the target region was then subtracted from the host time to obtain the overhead times. All measurements were taken on the TI K2 Rev 3.0 EVM using compilers and tools listed in § 4.2.5. Execution times were measured in microseconds averaged over 100 iterations with a standard deviation of less than 5% of the mean value.

Using the shared memory allocation API results in a maximum overhead of 3.75 ms for a 12288 KB (12 MB) buffer. Interestingly, the overhead of allocating larger buffers reduces gradually up to the largest benchmark size of 163840 KB (160 MB). Using `malloc` results in overheads increasing proportional to buffer size. Speed-up factors shown in Figure 4.10 suggest that for small buffer sizes up to 64 KB `malloc`'d buffers have almost equivalent performance compared to `__malloc_ddr`'d ones. However, larger buffers, especially ones above 1 MB should be allocated using `__malloc_ddr` as it results in a considerable performance advantage.

4.2.4 OpenMP runtime optimization: Utilizing target scratchpad memory

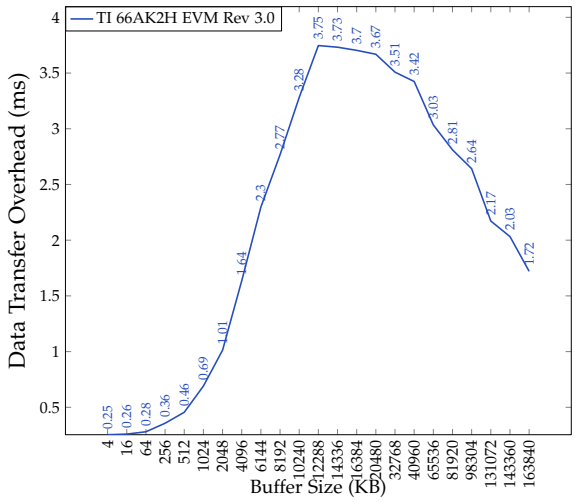
On the K2, each DSP core's 1MB L2 memory can be configured such that a portion of it is fast scratchpad memory and the rest is L2 cache. The OpenMP 4.0 map clause specification does not allow taking advantage of the scratchpad memory. In this section, a new map-type is proposed to reserve a chunk of fast scratchpad memory resident in the L2 SRAM portion of each K2 DSP core for usage during target region execution.

We added support for a new `local` map-type, which maps a buffer to the L2 scratchpad memory. In terms of data synchronization, such buffers are treated as map-type `alloc`. They have an undefined initial value on entry to the target region and any updates to the buffer in the target region cannot be reflected back to the host.

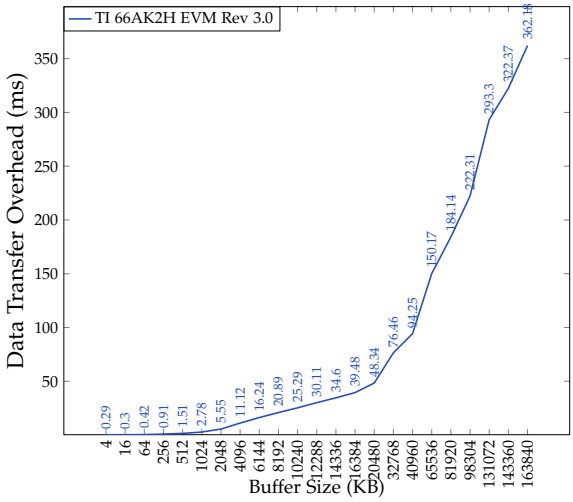
Given that the L2 SRAM has a read latency of less than 10 cycles compared to DDR read latency of 60-80 cycles, reserving and using L2 SRAM though a `local` map is expected to provide higher performance compared to simply using buffers in DDR.

4.2.5 Evaluation

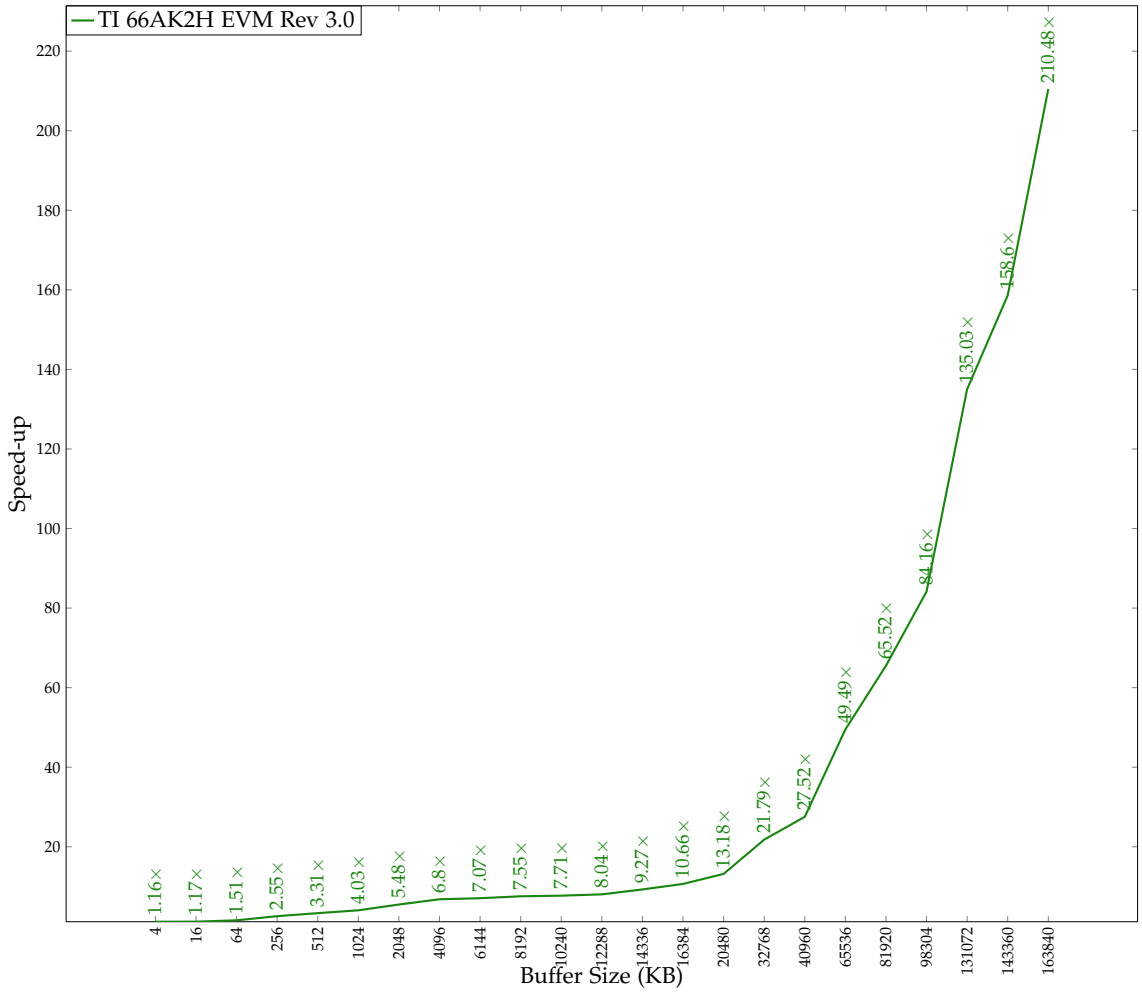
To quantify the overhead of using the OpenMP 4.0 runtime environment on the K2 SoC, we use the matrix multiplication (GEMM) application. Given that the performance of the K1 SoC was portrayed through use of GEMM [Igual et al.,



(a) `__malloc_ddr`



(b) `malloc`



(c) `Speed-up`

Figure 4.10: `__malloc_ddr()` speed-up vs. `malloc()`

2012] in previous work, it was a logical choice to evaluate performance of the K2. Fundamental to a majority of HPC applications are Basic Linear Algebra Subprograms (BLAS). For an emerging HPC platform, achieving good scalable performance for fundamental BLAS routines is a key factor.

```
1  double MatmulOptTarget(real* A, real* B, real* C,
2                         int m, int k, int n,
3                         real* MSMC_buf, int msmc_size, int num_threads)
4  {
5      int size_A = m*k; int size_B = k*n; int size_C = m*n;
6      /* Local L2 SRAM scratch total: 768 Kbytes = 786432 bytes */
7      /* Do not need to allocate here, passing a null pointer as local is adequate */
8      real *pL2 = NULL;
9      uint64_t elapsed_cycles;
10     #pragma omp target map(to: A[0:size_A], B[0:size_B], m, k, n, num_threads) \
11                        map(alloc: MSMC_buf[0:msmc_size]) \
12                        map(local: pL2[0:L2_SRAM_NUM_REALS]) \
13                        map(tofrom: C[0:size_C]) \
14                        map(from: elapsed_cycles)
15     {
16         omp_set_num_threads(num_threads);
17         /* Set start address of a core's L1D SRAM that is about to be created */
18         real* pL1 = (real*) L1D_SRAM_START;
19         int lda = m, ldb = k, ldc = CORE_PROCESS_ROWS*((m+(CORE_PROCESS_ROWS-1))/
               CORE_PROCESS_ROWS);
20         int tid, mLocal;
21         uint64_t start = __clock64();
22         #pragma omp parallel default(none) \
23                              private(tid, mLocal) \
24                              firstprivate(pL1, pL2, m, n, k, lda, ldb, ldc) \
25                              shared(A, B, C, MSMC_buf)
26         {
27             /* Configure L1D on each core to have 16KB SRAM and 16 KB Cache */
28             __cache_l1d_16k();
29             int nthreads = omp_get_num_threads();
30             int mRemaining = 0;
31             tid = omp_get_thread_num();
32             mLocal = (nthreads > m ? 1 : m/nthreads);
33             if (tid == nthreads-1)
34                 if (m % nthreads != 0) mRemaining = m % mLocal;
35             gemm(mLocal + mRemaining, n, k, A + mLocal*tid, lda, B, ldb, C + mLocal*tid,
                    ldc,
36                   pL1, pL2, MSMC_buf, tid);
37             /* Restore L1D Cache config on each core to entire 32KB Cache */
38             __cache_l1d_all();
39         }
40         uint64_t end = __clock64();
41         elapsed_cycles = end - start;
42     }
43     /* Assuming the clock speed of the DSP is 1.228Ghz Calculate the elapsed seconds */
44     double elapsed_time_sec = ((double)elapsed_cycles)/1.228e9;
45     return elapsed_time_sec;
46 }
```

Figure 4.11: Target region for GEMM

Several approaches have been used to perform blocking/paneling of matrices

to maximize utilization of caches and scratchpad RAM. In [Ali et al., 2012], GEMM was written for the C6678 DSP and the performance measured across 8 DSP cores was 79.4 GFLOPS for SGEMM and 21 GFLOPS for DGEMM with the DSPs running at 1Ghz. To implement GEMM for K2, the same algorithm, inner kernel and matrix paneling scheme was used. In order to maximize the use of a larger L2 and MSMC SRAM, the number of inner panels was increased to use all 768 KB of L2 and 4.5 MB of MSMC SRAM. The OpenCL runtime reserves 128 KB of L2 and 1.5 MB MSMC SRAM. The remaining 128 KB of L2 is configured as cache.

The GEMM implementation for the C6678 DSP Ali et al. [2012] used the EDMA3 co-processor for data transfer and intrinsics such as `_cmpysp()` to perform four single-precision floating-point multiplies and `_daddsp()` to perform four single-precision additions in one cycle.

The OpenMP 4.0 runtime environment requires that each function used within a target region must have an equivalent implementation for the host. The `if()` clause specification mandates this. However, it is important to note that every accelerator specific function or API might not be implementable for the host side. This situation was accounted for by allowing C66x specific code to be compiled separately and linked into the final executable. In this case the GEMM API function was compiled with instrumentation code and the inner kernel using the DSP compiler `cl6x` and called the GEMM function from within the target region as shown in Figure 4.11.

Timing within the target region was done using the built-in `__clock64()` function that provides a cycle count by reading a DSP performance counter register.

In Figure 4.11 several key optimizations for using the K2 memory hierarchy are shown. The *local* map-type is used to allocate a 768 KB buffer on each DSP core's L2 SRAM. Upon entering the target region, the L1D cache size is halved to create 16 KB L1D SRAM and 16 KB L1D Cache per DSP core. This optimization is critical to the performance of the particular paneling scheme used. It ensures that a 16KB panel is retained in L1D SRAM as long as possible while the other 16KB cache usage is maximized using the `__touch(void* array, int size)` built-in function that allows fast reading of a memory segment into cache.

The arrays A, B and C were allocated in shared memory using `__malloc_ddr()`. The `MSMC_buf` array used 4MB of usable 4.5 MB in MSMC SRAM and was allocated using `__malloc_msmc()`. Within the instrumentation code, the EDMA Manager API was used to start parallel DMA transfers on separate channels. Figure 4.12 shows the API function for a 2D block transfer which allows trans-

ferring a number of lines of contiguous chunks of memory with different source and destination start offsets (pitch).

```
1 int32_t EdmaMgr_copy2D2DSep( EdmaMgr_Handle h,void *restrict src, void *restrict dst,
2                              int32_t  num_bytes, int32_t  num_lines,
3                              int32_t  src_pitch, int32_t  dst_pitch);
```

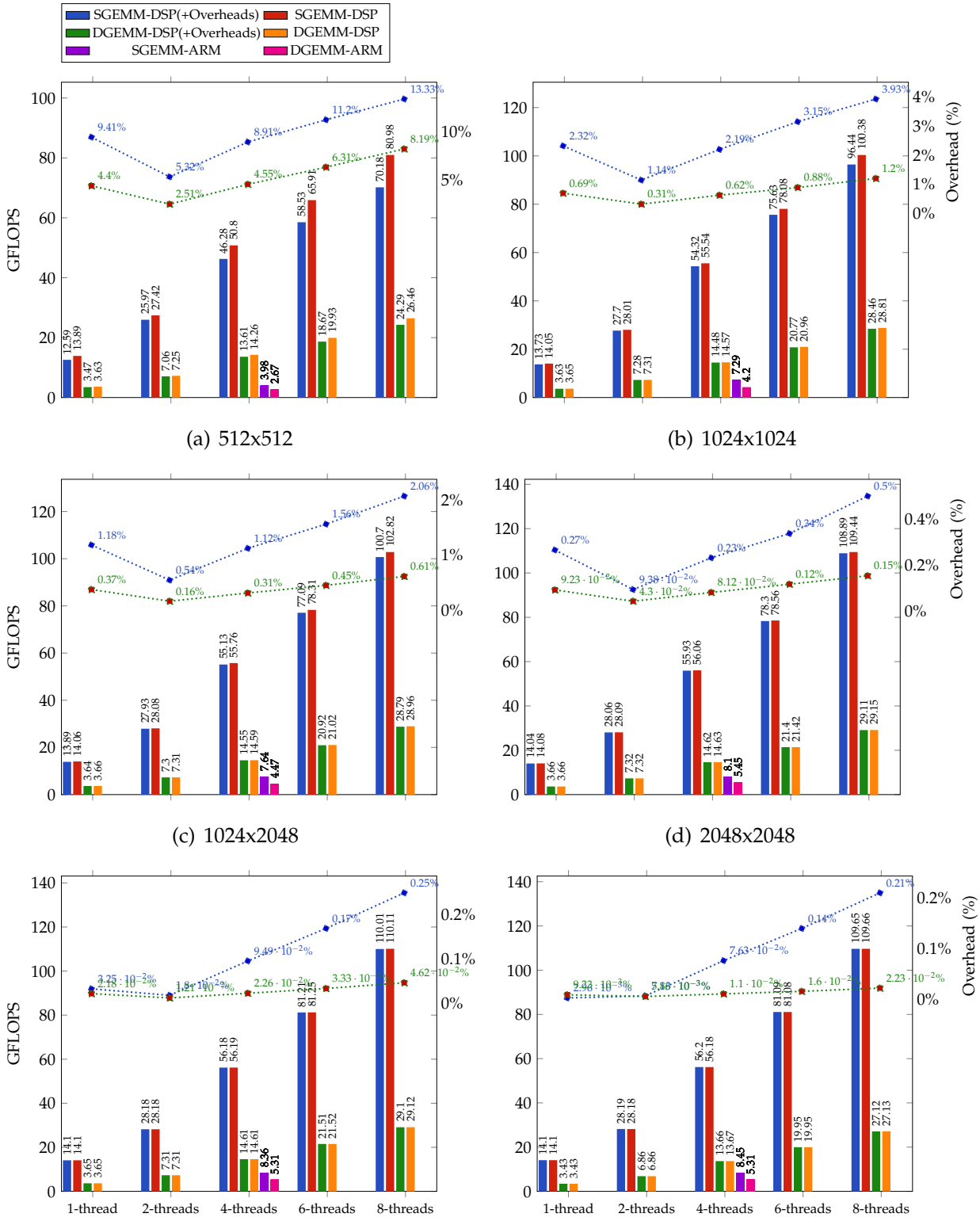
Figure 4.12: EDMA Manager 2D transfer API

4.2.5.1 Performance

Performance was measured on a TI K2 EVM (Rev 3.0). The ARM Cortex-A15 cores were clocked at 1.2 Ghz and the C66x DSP cores ran at 1.228 Ghz. In order to assess GEMM performance on the ARM cores, ATLAS CBLAS (using Pthreads) v3.10.1 library auto-tuned for ARM Cortex-A15 was used. The auto-tuning was performed on the ARM cores using gcc 4.7.2. The following TI software packages were used: TI MCSDK Linux v3.00.04.18, TI OpenCL v0.10.0, OpenMP accelerator model v1.0.0 and TI C6000 Code Generation Tools v8.0.0.

Two sets of times were collected. The first set was measured from within the target region using `__clock64()` to time the OpenMP parallel region and DSP execution time. GFLOPS calculated using this time are denoted as SGEMM-DSP and DGEMM-DSP in Figure 4.13. The second set was measured from the host using the `clock_gettime()` function call and the `CLOCK_MONOTONIC_RAW` clock with microsecond accuracy. This time measurement included the overhead of calling the accelerator model runtime library and the data transfer time to and from the target region along with execution time. SGEMM-DSP(+Overheads) and DGEMM-DSP(+Overheads) denote GFLOPS calculated using time measured from the host. `clock_gettime()` was also used to measure time for ATLAS Pthread CBLAS SGEMM and DGEMM on ARM. SGEMM-ARM and DGEMM-ARM denote GFLOPS for these measurements. Each time measurement was averaged over a 100 iterations with a standard deviation of less than 5% of the mean value. GFLOPS values calculated using these time measurements are reported in Figure 4.13.

Consider first the overhead of using the OpenMP 4.0 runtime environment. Figure 4.13 reports on the overheads for each experiment run. The overhead is calculated as the difference between the time measured on the ARM host vs the time measured on the DSP as a fraction of the host time. The dotted lines display the overhead percentages, with the blue dotted line representing overheads for SGEMM experiments and the green dotted line representing DGEMM.

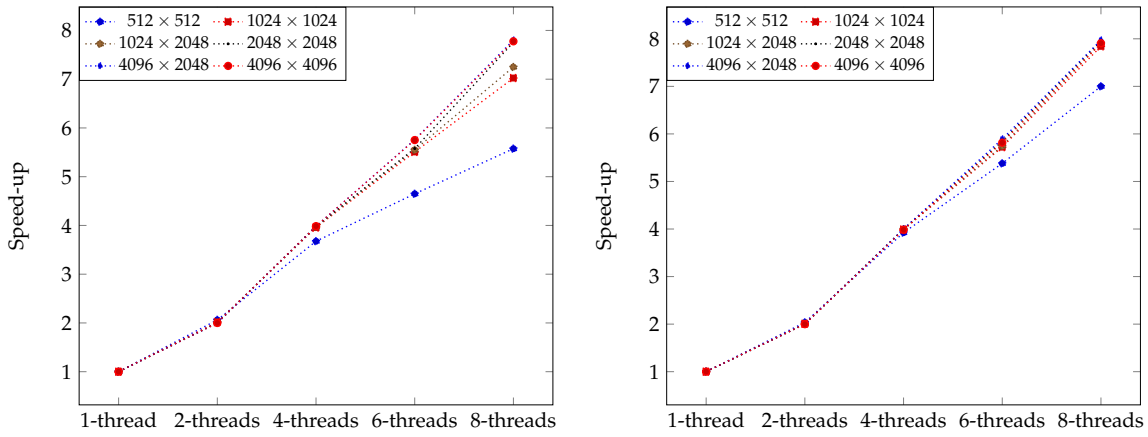


(a) 512x512 (b) 1024x1024

(c) 1024x2048 (d) 2048x2048

(e) 4096x2048 (f) 4096x4096

Figure 4.13: GEMM Performance on Keystone II using OpenMP 4.0 runtime environment



(a) SGEMM (b) DGEMM

Figure 4.14: DSP GEMM performance scaling

Consider the smallest 512 × 512 matrices. Overheads reported for SGEMM measurements, between 5.32% to 13.33%, are higher than those reported for DGEMM, between 2.51% to 8.19%. Compared to measurements for larger matrix sizes, these overheads are significantly higher. This is because an extra five buffers are mapped to the target region for execution and the DSP execution times for smaller cases do not outweigh the cost of creating these buffers.

For larger matrices such as the 4096 × 2048 and 4096 × 4096, the overheads appear negligible and in the order of 0.01%. This implies that the larger the problem size, the lower the overhead fraction, which further implies that overhead is largely constant.

Consider now single thread performance of the DSP cores via the OpenMP 4.0 runtime environment. Peak performance for SGEMM is measured at 14.1 GFLOPS for the square 4096 × 4096 matrices. Peak performance for DGEMM was measured at 3.66 GFLOPS for the square 2048 × 2048 case. SGEMM performance was 3.85× better than DGEMM. This is expected given that the DSP cores are 32-bit and do not have native double-precision instructions which must be simulated using multiple single-precision instructions.

Consider the performance of ATLAS auto-tuned GEMM on the ARM cores. ATLAS was tuned to run only for the 4-thread configuration and therefore other thread configurations were not performed. The peak SGEMM performance across four ARM cores was 8.45 GFLOPS in the largest case and for DGEMM it was 5.45 GFLOPS for the 2048 × 2048 case. Compared to 4-thread performance on the DSP cores, ATLAS on ARM was 6.7× slower for SGEMM and 2.6× slower for DGEMM.

Consider now the scaling across different thread configurations. The Figure 4.13 reports strong scaling performance fixed total problem sizes. The number of DSP threads used is varied and each DSP core runs a single thread. Figure 4.14 reports the speed-ups measured when adding more DSP threads. Near linear scaling is observed in all cases except the smallest 512×512 case for both SGEMM and DGEMM experiments.

It is also evident that even for rectangular matrices performance is not diminished. In fact, the rectangular 4096×2048 matrix case provides the peak performance of 110.11 GFLOPS for SGEMM across all eight DSP cores. For DGEMM the square 2048×2048 case provides the peak DGEMM performance of 29.15 GFLOPS.

With respect to the peak performance of the DSP cores, using the OpenMP 4.0 runtime environment on K2 achieves 70.05% efficiency for SGEMM and 74.18% efficiency for DGEMM including overheads. The original implementation for C6678 achieved 62% for SGEMM and 65% for DGEMM [Ali et al., 2012]. The increase in efficiency can be attributed to larger L2 and MSMC SRAM available on the K2.

4.3 Summary

Evaluation of a bare-metal OpenMP runtime on the K2 DSP indicated that the absence of hardware cache coherence was the primary performance limitation as a significant computational cost of 1350 cycles was paid at each OpenMP synchronization point. Minimization of the number of such synchronization points is recommended for all application codes. Loops in OpenMP must also be scheduled using larger block sizes for each DSP core for better performance.

The design, implementation and evaluation of an OpenMP 4.0 runtime for the K2 SoC was presented. An OpenMP 4.0 to OpenCL mapping was defined and implemented in the `libOpenMPAcc` host library. It is possible to use this mapping on any other LPSoC platform. A meta-compiler tool called `clacc` was implemented to integrate all stages of transforming an OpenMP 4.0 source code to a fat binary executable on the K2.

The OpenMP 4.0 runtime was evaluated on a TI K2 Rev 3.0 EVM using an implementation of the GEMM application. SGEMM performance of 110.11 GFLOPS and DGEMM performance of 29.15 GFLOPS was achieved. Overheads of the runtime were measured to be negligible for larger problem sizes.

Performance of GEMM on 4 DSP cores was measured to be 6.7× faster for SGEMM and 2.6× faster for DGEMM compared to 4 ARM cores. Using 8 DSP

cores was at least 12× faster for SGEMM and 5× faster for DGEMM compared to 4 ARM cores. Near linear performance scaling was also observed for larger matrix sizes computed across multiple DSP cores.

Better performance efficiency of 74.18% was achieved on K2 DSP cores compared to K1 DSP cores. This indicated that performance was maintained when offloading from the ARM Cores to the DSP cores. The improvement in efficiency was due to extra L2 and MSMC scratch memory on K2.

The OpenMP 4.0 runtime was extended to provide the `__malloc_ddr()` and `__malloc_msmc()` functions designed to dynamically allocate memory from the host such that it is in contiguous and shared physical memory, therefore avoiding any copies when mapping the memory to the target device.

A new `local` map-type to allocate a buffer on DSP L2 SRAM was introduced. As evidenced in the GEMM implementation, high performance can be achieved when using local memory in L2 SRAM.

Contributions from work described in this chapter have enabled use of the K2 SoC for HPC workloads using the OpenMP 4.0 programming model. Using our implementation of an OpenMP 4.0 runtime, it is possible to offload computation from the ARM cores to the DSP cores on K2. To our knowledge, this is the first implementation of an OpenMP 4.0 runtime on any LPSoC system.

In summary, with its combination of ARM Cortex-A15 and TI C66x DSP cores integrated on a single SoC with shared memory among all the cores, the TI K2 architecture is a good match for OpenMP 4.0. The performance optimizations described for the OpenMP runtime are general enough to be considered for future extensions of the OpenMP specification.

In the next chapter, a novel communication framework is described that allows both the K1 and K2 SoC's present on the nCore BrownDwarf system to be programmed simultaneously.

Chapter 5

Exploiting multiple Keystone SoCs on BrownDwarf

Chapter 4 considered the suitability of the OpenMP programming model on a single Keystone II SoC with the design and implementation of an OpenMP 4.0 runtime environment to offload computation from the ARM to the C66x DSP cores. Work described in this chapter considers offloading computation from a Keystone II (K2) SoC to a Keystone I (K1) SoC on the nCore BrownDwarf system via a new message passing communication framework. The BrownDwarf system has a unique architecture composed of multiple Keystone SoCs connected via the Hyperlink interface within each node. In order to effectively use a single BrownDwarf node, both K1 and K2 SoCs should be executing computation simultaneously.

The primary objective of work presented in this chapter is to design, implement and evaluate a functional and proof-of-concept programming environment that enables the simultaneous use of all ARM and DSP cores present on both the K2 and K1 SoCs across multiple nodes of a BrownDwarf system. A secondary objective is to identify bottlenecks in the system for future optimization.

When BrownDwarf hardware first became available to HPC developers in 2013, it was only possible to program its various SoCs individually. Both K1 and K2 DSP cores could be used with TI's bare-metal OpenMP runtime while the K2 DSP cores also had the option of being programmed with OpenCL. Various options were available for the ARM cores including GNU OpenMP, Pthreads and C++ 11 threads.

A low-level driver called the Multi-process Manager (MPM) was provided by TI. MPM could be used to read and write data packets across Hyperlink. However, no higher level framework was available which an application could use to offload computation from the ARM cores on the K2 to the DSP cores on K1 SoCs.

This chapter describes the design and implementation of such a communi-

cation framework. It enables an application to work with both OpenCL and OpenMP 4.0 to launch kernels from K2 ARM cores to K2 DSP cores alongside offloading code using message passing communication to the K1 DSP cores. To the best of our knowledge, this is the first implementation of a message passing communication framework across the Hyperlink interface allowing multiple Keystone SoCs to work simultaneously.

To co-ordinate between both SoCs, the communication framework is designed with two primary components. A *host* side library called *libk1comms* running on the K2 ARM cores and a *target* side library called *k1dspmonitor* running on the K1 DSP cores. Libk1comms orchestrates message passing to the k1dspmonitor which runs continuously looking for and handling work messages from the K2. The bare-metal OpenMP runtime described in § 4.1 was linked into the k1dspmonitor since an OpenCL runtime was not available for K1 DSP cores. Communication between libk1comms and k1dspmonitor is enabled using a new data transfer API using Hyperlink.

Section 5.1 describes the implementation of a new data transfer API for Hyperlink and the evaluation of its network characteristics. Section 5.2 describes the high and low level design of the new communication framework while section 5.3 describes the associated application build system. Section 5.4 describes an evaluation of the communication framework using GEMM matrix multiplication across all processing elements on four BrownDwarf nodes.

5.1 A Data Transfer API for Hyperlink

The objective of work presented in this section was to provide a means to transfer data buffers of arbitrary length between K2 and K1 SoCs across Hyperlink on a single BrownDwarf node using DMA co-processors on the K2 SoC. A new high-level data transfer API was implemented using the low-level MPM transport library. This API is eventually used by the communication framework for data transfer.

The MPM API is divided into two categories. The first category deals with data packet transactions between ARM virtual memory on the K2 and DSP physical memory on the K1. The second deals with use of the Enhanced DMA (EDMA) engine present on the K2 to perform data transactions as shown in Figure 2.8 in § 2. In order to use MPM DMA APIs, a chunk of contiguous memory was reserved in the ARM address space which could then be used as a staging area for transactions between ARM virtual memory on K2 and DSP physical memory on K1.

Since our objective of transferring arbitrary sized buffers could be achieved by breaking up the buffer into multiple smaller chunks to be transferred separately, reservation of a large amount of memory was not required for MPM DMA. On the BrownDwarf system, a 64MB chunk of DMAMEM was reserved at the Linux kernel device tree level with start address at $0x81c000000$ on K2 DRAM. ARM Linux and DSP CMEM used the rest of the K2 DRAM. This ensured that Linux would not consider this section of DRAM as part of its main memory and this could be directly addressed after an `mmap()` system call to map that contiguous chunk to a pointer in virtual memory enabling ARM cores to use it.

Figure 5.1 presents the data transfer API implemented for use on ARM cores. The API considers ARM DRAM as local memory and K1 DRAM as remote memory. The `hlink_read()` function can be used to read a buffer of size up to 4MB from remote memory. `hlink_write()` can write a buffer of up to 4MB to remote memory. Since the BrownDwarf Hyperlink interface is configured with 64 memory regions of 4MB each, a read/write operation cannot be performed on buffers larger than 4MB. These two functions do not use MPM DMA operations and can therefore be used to transact with the lower 4GB of K1 memory (32-bit address space). The MPM APIs used by these functions are `mpm_transport_read()/mpm_transport_write()`. These calls are intentionally implemented to not support larger buffers than 4MB since the DMA based API is intended to handle those transfers.

Using the MPM DMA API calls, the `hlink_dma_read()` function can be used to read a buffer of arbitrary size from remote memory. `hlink_write()` can similarly write a buffer of arbitrary size to remote memory. The MPM DMA API calls used are `mpm_transport_get_initiate64()` and `mpm_transport_put_initiate64()`. Such DMA initiated transactions have stages involving the MPAX unit as described in Figure 2.8 in § 2 and are therefore suitable for larger buffers than 4MB.

Consider the implementation of one of API functions, `hlink_dma_write()` given in Figure 5.2. The first 16MB of reserved DMAMEM starting at `DMA_RX_BUF` is reserved for Hyperlink read operations. The next 16MB starting at `DMA_TX_BUF` is reserved for Hyperlink write operations.

Since there are multiple DMA channels available for use on the K2 SoC, a decision was made to use 4 channels to at least enable two parallel read and two parallel write transactions. In this scenario, each of the 4 ARM cores could initiate a parallel transaction to K1 memory using DMA channels. Since the transactions are synchronous, more than 4 threads running across 4 ARM cores would start multiplexing between each other leading to transfer delays. The

```
1  /* Open Hyperlink MPM transport */
2  void hlink_open(char* transport, mpm_transport_open_t* ocfg,
       mpm_transport_h* mpmh);
3
4  /* Read from Hyperlink MPM transport */
5  void hlink_read(uint64_t memory_region,
6                  uint32_t length,
7                  char* read_buffer,
8                  mpm_transport_read_t* rcfg,
9                  mpm_transport_h* mpmh
10                 );
11
12 /* Read from Hyperlink MPM transport */
13 void hlink_write(uint64_t memory_region,
14                  uint32_t length,
15                  char* write_buffer,
16                  mpm_transport_write_t* wcfg,
17                  mpm_transport_h* mpmh
18                  );
19
20 /* Read from Hyperlink MPM transport using EDMA */
21 void hlink_dma_read(uint64_t memory_region,
22                     uint32_t length,
23                     char* read_buffer,
24                     mpm_transport_read_t* rcfg,
25                     mpm_transport_h* mpmh
26                     );
27
28 /* Write to Hyperlink MPM transport using EDMA */
29 void hlink_dma_write(uint64_t memory_region,
30                      uint32_t length,
31                      char* write_buffer,
32                      mpm_transport_write_t* wcfg,
33                      mpm_transport_h* mpmh
34                      );
35
36 /* Close Hyperlink MPM transport */
37 void hlink_close(mpm_transport_h* mpmh);
```

Figure 5.1: Hyperlink Data Transfer API

higher 32MB of DMAMEM is used for the parallel set of read/write operations.

The first step in the DMA write operation is to memory map the 16MB DMA TX region to a buffer in ARM virtual memory given in line 20. A while loop is then initiated which operates until all bytes of the source buffer are transferred. This loop iterates over chunks of `DMA_TX_BUF_SIZE` or smaller. In this case, the chunk size is 16MB. After some initial experimentation with different buffer sizes, using a 16MB buffer was found to have the highest performance efficiency. Each such chunk is first copied to the memory mapped buffer in DMAMEM as shown in lines 30 and 43. A helper function `hlink_block_dma_write()` shown in lines 32 and 45 is then called to complete the transfer of this 16MB or smaller chunk.

Figure 5.4 shows the `hlink_block_dma_write()` helper function. This function is used to perform transfers of buffers sized `DMA_TX_BUF_SIZE` or smaller using blocks of 4MB or smaller. The actual size of a transfer block depends on its position with respect to a 4MB Hyperlink segment boundary. Since Hyperlink is configured with 64 segments of 4MB each, transfers of 4MB blocks must not overlap two consecutive segments. Therefore, a 4MB block transfer is only possible if the destination address corresponds with a 4MB hyperlink segment boundary. For blocks with non-aligned destination addresses, the transfer size must be adapted.

Another helper function `get_hlink_offset()` shown in Figure 5.3 was implemented to calculate a block size limit to contain a transfer within a 4MB Hyperlink segment. This function is used in line 17 to calculate the effective data transfer size before an `mpm_transport_put_initiate64()` operation is performed.

A while loop then iterates over blocks of `HLINK_SEGLEN_BYTES` or smaller and transfers them using the EDMA write operation described in Figure 2.8.

5.1.1 Performance Evaluation

A set of latency and bandwidth benchmarks were performed using our Hyperlink data transfer API to evaluate its performance. Table 5.1 presents results from these benchmarks. For each benchmark a data buffer in K2 DDR3 memory was used to initiate a read or write operation from the ARM to a remote DDR3 memory of a K1 SoC. Each benchmark was repeated a hundred times and the result was averaged across all runs. The `clock_gettime()` timing function was used along with the `CLOCK_MONOTONIC` timer.

Consider first the latency benchmark. A small 32 byte buffer was used to perform each data transfer and its time taken was recorded. Each of the latency

```
#define MAP_SIZE_16MB       0x1000000
#define DMAMEM_START        (0x81c000000) /* Start  of 64MB DMAMEM */
#define DMA_RX_BUF          (DMAMEM_START)
#define DMA_TX_BUF          (DMA_RX_BUF + MAP_SIZE_16MB)
#define DMA_RX_BUF_SIZE     MAP_SIZE_16MB
#define DMA_TX_BUF_SIZE     MAP_SIZE_16MB

/* Write to Hyperlink MPM transport using EDMA */
void hlink_dma_write(uint64_t memory_region,
                     uint32_t length,
                     char *write_buffer,
                     mpm_transport_write_t *wcfg,
                     mpm_transport_h *mpmh
                    )
{
    char *buffer = NULL;
    int fdm, ret = 0;
    /* MMAP a region in contiguous DMA memory */
    fdm = open("/dev/mem", O_RDWR | O_SYNC);
    buffer = (char *) mmap64(NULL, DMA_TX_BUF_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
        fdm, DMA_TX_BUF);
    uint64_t offset = 0;
    uint32_t remaining_bytes = length;

    while (remaining_bytes > 0)
    {
        /* If the transfer size is less than the DMA_TX_BUF_SIZE, then continue with
            chunking of DMA blocks */
        if (remaining_bytes < DMA_TX_BUF_SIZE)
        {
            /* Copy remaining data from write_buffer into mmap'd buffer */
            memcpy(buffer, write_buffer + offset, remaining_bytes);
            /* Write from mmap'd buffer into Hyperlink in chunks at a time */
            ret = hlink_block_dma_write(memory_region + offset,
                                        remaining_bytes,
                                        DMA_TX_BUF,
                                        wcfg,
                                        mpmh
                                       );
            remaining_bytes = 0;
        }
        else
        {
            /* Copy data from write_buffer into mmap'd buffer upto DMA_TX_BUF_SIZE*/
            memcpy(buffer, write_buffer + offset, DMA_TX_BUF_SIZE);
            /* Write from mmap'd buffer into Hyperlink in chunks of at a time */
            ret = hlink_block_dma_write(memory_region + offset,
                                        DMA_TX_BUF_SIZE,
                                        DMA_TX_BUF,
                                        wcfg,
                                        mpmh
                                       );

            offset += DMA_TX_BUF_SIZE;
            remaining_bytes -= DMA_TX_BUF_SIZE;
        }
    }

    /* Cleanup */
    if (buffer) munmap(buffer, DMA_TX_BUF_SIZE);
    close(fdm);
}
```

Figure 5.2: Hyperlink DMA transfer

```
#define HLINK_SEGLEN_BYTES  0x400000 /* 4MB */
#define HLINK_MSMC_BASE     0x0c000000
#define HLINK_DDR_BASE      0x80000000
uint64_t get_hlink_offset(uint64_t memory_region)
{
    uint64_t base_diff;
    if (memory_region > HLINK_DDR_BASE) base_diff = memory_region - HLINK_DDR_BASE;
    else                                base_diff = memory_region - HLINK_MSMC_BASE;
    return (base_diff % HLINK_SEGLEN_BYTES);
}
```

Figure 5.3: Hyperlink DMA block offset

benchmarks was a one-way data transfer, either read from remote memory or write to remote memory. Time was measured for an entire blocking data transfer to complete i.e. including the overhead of the entire software stack underneath. Time taken to initialize, open and close the Hyperlink interface was not included.

A latency of 27μs was recorded for each API call. A maximum read bandwidth of 89.78 MB/s and write bandwidth of 539.29 MB/s was measured for a 4MB buffer.

The end-to-end latency of 10Gb Ethernet link for a TCP/IP packet is measured to be 19μs [Feng et al., 2005] for a 32 byte transfer. Given that 10Gb Ethernet is an industry standard commonly used in low-latency bound application, and included in BrownDwarf system, it provides a suitable baseline measurement to compare Hyperlink against.

Compared to 10Gb Ethernet, Hyperlink latency is marginally slower but in the same order of magnitude. Given that the latency of non-DMA operations is equal to that of doing DMA operations, either of the operations can be used based on expected bandwidth.

Consider the 4MB bandwidth benchmark. Several 4MB buffers are used to perform each data transfer. The total duration is recorded and used to calculate the effective transfer speed in MB/s. The combined peak theoretical bandwidth of the link is 2.78 GB/s.

We observe that Hyperlink write bandwidth is considerably higher than read bandwidth. Non-DMA write bandwidth is measured at 539.29 MB/s and is 17.29$\times$ faster than read bandwidth measured at 31.19 MB/s. These transfers do not saturate the Hyperlink lanes. The write speed is only 18.9% of peak and the read speed is 1.1% of peak bandwidth. Using DMA, the write bandwidth reduces. This is due to the extra memory copy operation from ARM virtual memory to contiguous DMAMEM before DMA can be initiated. DMA write is

```
/* Write from mmap'd buffer into Hyperlink in chunks at a time */
int hlink_block_dma_write(uint64_t memory_region,
                          uint32_t length,
                          uint64_t dma_tx_buf_addr,
                          mpm_transport_write_t* wcfg,
                          mpm_transport_h* mpmh
                          )

{
    mpm_transport_trans_h t;
    uint32_t remaining_bytes = length;
    uint64_t offset = 0;
    uint32_t block_limit;

    while (remaining_bytes > 0)
    {
        block_limit = (uint32_t)(HLINK_SEGLEN_BYTES - get_hlink_offset(memory_region +
            offset));

        if (remaining_bytes < block_limit)
        {
            t = mpm_transport_put_initiate64(*mpmh,                    /* Handle      */
                                             memory_region + offset,   /* Dest Addr   */
                                             dma_tx_buf_addr + offset, /* Source Addr */
                                             remaining_bytes,          /* Length      */
                                             true,                     /* is_blocking */
                                             wcfg                      /* Write Config*/
                                             );
            remaining_bytes = 0;
        }
        else
        {
            t = mpm_transport_put_initiate64(*mpmh,                    /* Handle      */
                                             memory_region + offset,   /* Dest Addr   */
                                             dma_tx_buf_addr + offset, /* Source Addr */
                                             block_limit,              /* Length      */
                                             true,                     /* is_blocking */
                                             wcfg                      /* Write Config*/
                                             );

            offset += block_limit;
            remaining_bytes -= block_limit;
        }
    }

    return 0;
}
```

Figure 5.4: Hyperlink DMA block'd transfer

Data Transfer API	Latency-32b (μs)	Bandwidth-4MB (MB/s)	Bandwidth-16MB (MB/s)
hlink_write()	27	539.29	-
hlink_read()	27	31.19	-
hlink_dma_write()	27	335.72	361.42
hlink_dma_read()	27	89.78	90.38

Table 5.1: Hyperlink data transfer speed

measured at 335.72 MB/s and is 1.6× slower than non-DMA write. DMA read however is 3× faster than non-DMA read.

Consider the 16MB bandwidth benchmark. Similar to the 4MB benchmark, this is also performed with multiple 16MB buffer transfers. Since we only provide blocked DMA transfers for sizes above 4MB, this benchmark is only performed using our DMA API.

We observe a marginal increase in DMA write bandwidth compared to 4MB buffers. This is because the 16MB memory copy to DMAMEM is performed in a single operation in this benchmark and 4MB chunks are then transferred from DMAMEM to remote memory. To transfer a similar 16MB data buffer using the 4MB benchmark, four separate memory copies are required which adds overhead.

It is evident that both the read and write bandwidth measurements across Hyperlink indicate a performance bottleneck. While using larger data sizes might increase the throughput, the configuration of Hyperlink on BrownDwarf with 4MB transfer windows limits performance. To our knowledge, this is the first study evaluating data transfer rates across Hyperlink on an HPC system. Further analysis and optimization of performance bottlenecks identified in this study would be of significant interest in future work.

5.2 Communication Framework Design

In this section we describe the framework developed to offload computation from the K2 ARM processor to the K1 DSP processors and transfer data between their respective DDR and MSMC SRAM memory regions.

This framework must be usable in the same application alongside the K2 OpenCL, OpenMP 4.0 runtimes and MPI or other communication libraries across several BrownDwarf nodes. Another necessary requirement was to support either OpenCL or OpenMP programming models on the K1 in order to enable their use with HPC application codes.

Consider the communication framework components in the context of an application running on the BrownDwarf. Figure 5.5 describes how the communi-

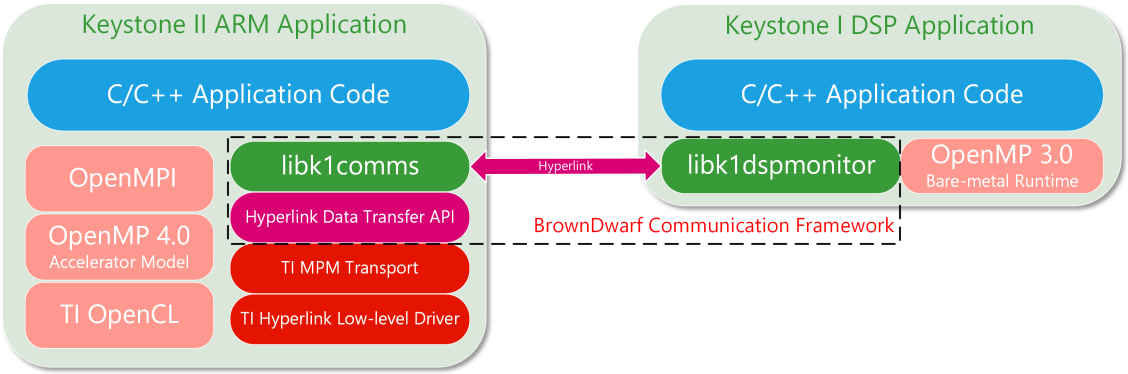


Figure 5.5: BrownDwarf Application Software Stack

cation framework components fit into the software stack used both on the ARM and DSP. A C/C++ application on the ARM would link against libk1comms and use its API to control the K1 DSP cores. Similarly, the corresponding libk1dspmonitor would be linked against by the DSP side of the application.

A decision was made to pre-define and therefore pre-compile the computation kernels to be executed on the K1 DSP cores. Since only OpenMP 3.0 C code was supported on the K1, pre-compiling OpenMP kernels was a natural choice. Further, this was chosen to remove the need for Just-In-Time (JIT) compilation and in the expectation that the user is able to define the K1 workload before execution.

5.2.1 Message Passing Communication Protocol

The communication protocol is designed to support the sending of control messages from the K2 ARM cores to the K1 DSP cores and receiving of result and success/failure messages from the DSP to the ARM cores. The following messages are defined:

- (K2 ARM $\longleftrightarrow$ K1 DSP): Synchronize execution. Send a SYNC + ACK combination to synchronize executions on both ARM and DSP cores.
- (K2 ARM $\longleftrightarrow$ K1 DSP): Terminate DSP Monitor. Clean out heap, and close DSP monitor. Send and ACK from DSP to ARM to acknowledge termination.
- (K2 ARM $\rightarrow$ K1 DSP): Perform a work item as defined in the work message *kind*. After the work item is complete, the DSP writes back a result in a

separate mailbox while sending an ACK signal to confirm completion. The following kinds of work messages are supported.

  - `MSG_CREATE_BUF_DDR`, `MSG_CREATE_BUF_MSMC`: Buffer allocation in DDR or MSMC heaps
  - `MSG_FREE_BUF_DDR`, `MSG_FREE_BUF_MSMC`: Buffer de-allocation in DDR/MSMC heaps
  - `MSG_CACHE_INV`: Invalidate and write-back DSP core L1 and L2 caches
  - `MSG_START_FN`: Begin execution of kernel
  - `MSG_RESET`: Reset the DSP heap and mailbox memory

- (K1 DSP $\rightarrow$ K2 ARM): Result messages supported by the DSP in response to work messages from the ARM.

  - `MSG_CREATE_BUF_SUCCESS`: Confirmation of buffer allocation success along with buffer start address
  - `MSG_CREATE_BUF_FAIL`: Confirmation of buffer allocation failure

5.2.2 Use of mailbox communication

Although, the medium of communication is Hyperlink, the landing area of communication data transmitted via Hyperlink was defined in MSMC SRAM memory on the K1 SoC.

There are two reasons for using MSMC SRAM instead of standard DDR RAM as a communication staging area. First, MSMC SRAM is faster and has a 10-20 cycle access time compared to 80-100 cycle access time for DDR RAM from K1 DSP cores. Second, portions of MSMC SRAM can be declared non-cached with respect to DSP cores which alleviates the need for cache invalidation, write-backs and prevents false sharing. This provides the DSP cores faster access to data updated remotely through Hyperlink.

We define a set of *mailbox* segments in non-cached K1 MSMC SRAM. We use a single page (4096 bytes) of memory and create 32 mailbox segments of 128 bytes each in that page. The size of each mailbox segment is set at 128 bytes since that is the length of a DSP L2 cache line and therefore would be fetched in a single attempt when accessed.

Two kinds of mailboxes are defined. The first kind is called `Mbox` and is used for control signals and encapsulates a single 32-bit counter. The second kind is called `Data_Mbox` and is used for sending workload data signals across.

It encapsulates three 32-bit counters, one for control and two for other payload items such as memory addresses or work messages.

A single communication *channel* is defined between the K2 ARM and a K1 SoC. For each channel, four mailboxes are defined. Two `Mbox` segments for transmission (TX) and reception (RX) of control signals. Another two `Data_Mbox` segments for TX and RX of data signals. A TX mailbox with respect to the ARM corresponds to the RX mailbox with respect to the DSP and vice versa.

5.2.3 Keystone II ARM Library: libk1comms

The software design of the libk1comms ARM library is described in Figure 5.6. Consider first the base mailbox layer in this design. The `Mbox` class encapsulates the `udma_atom` structure which holds a single 32-bit counter. The `Data_Mbox` class encapsulates the `udma_atom_msg` structure which holds a three 32-bit values, one message kind and two payload values.

The `CommsMbox` class initializes and provides means to use the four mailboxes required to communicate across a single channel. It provides all the remote work functions described in the communication protocol such as allocation and de-allocation of buffers, invoking a kernel execution and terminating the K1 DSP instance.

It also utilizes the Hyperlink Data Transfer API we implemented to perform data transfers to and from K1 MSMC SRAM and DDR RAM. The `copy_to` (using `hlink_write`) and `copy_from` (using `hlink_read`) operations transfer data of size less than 4MB to/from buffers allocated on K2 memory (MSMC or DDR) to/from K1 memory (MSMC or DDR). The corresponding `dma_to` (using `hlink_dma_write`) and `dma_from` (using `hlink_dma_write`) operations using EDMA can transfer buffers of arbitrary length. These API calls are demonstrated in Figure 5.7.

The `CommChannel` class represents a channel between a K2 ARM core and K1 SoC using a specific MPM transport channel. It instantiates a `CommsMbox` object and uses it to communicate with the K1 DSP cores.

5.2.4 Keystone I DSP Library: k1dspmonitor

The k1dspmonitor library is essentially a simple finite state machine. Once initiated by libk1comms on the ARM, it initializes the data structure required for read and writing control signals and data messages to mailbox segments corresponding to those setup by libk1comms on K1 MSMC SRAM.

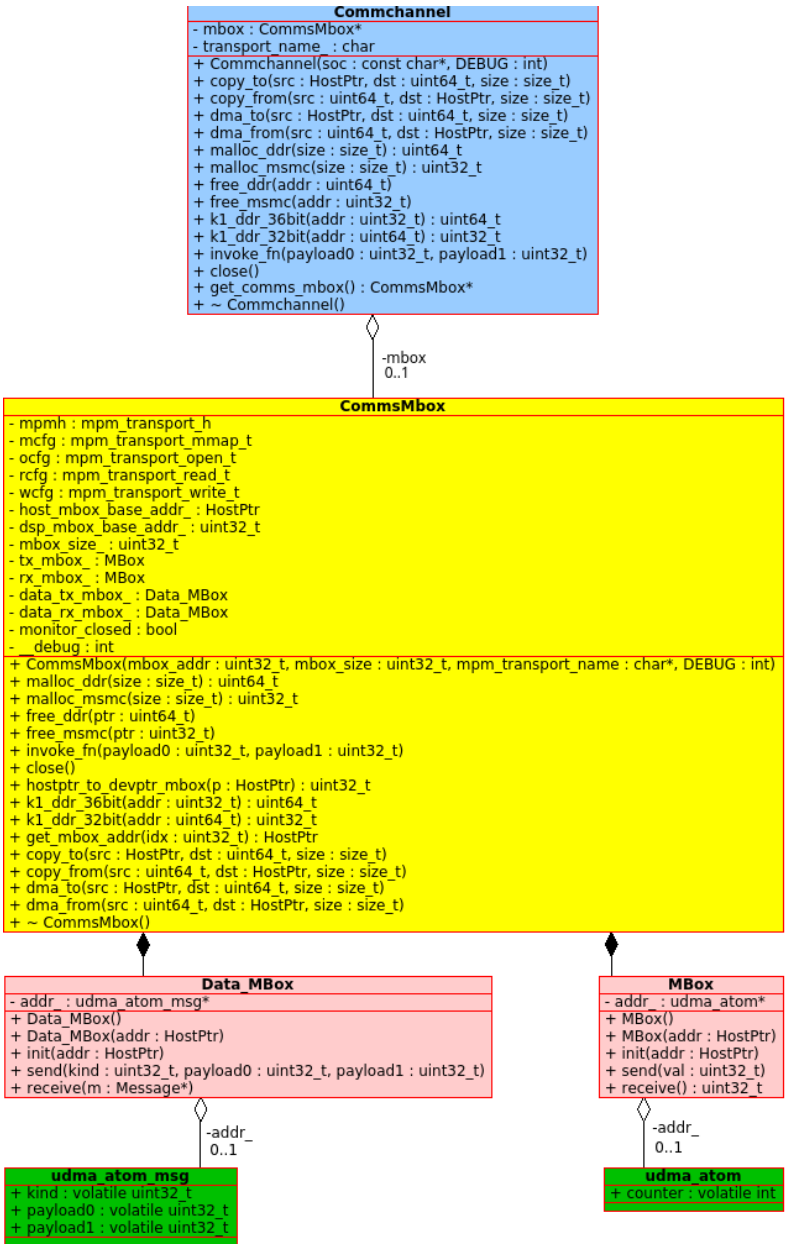


Figure 5.6: Software design of libk1comms

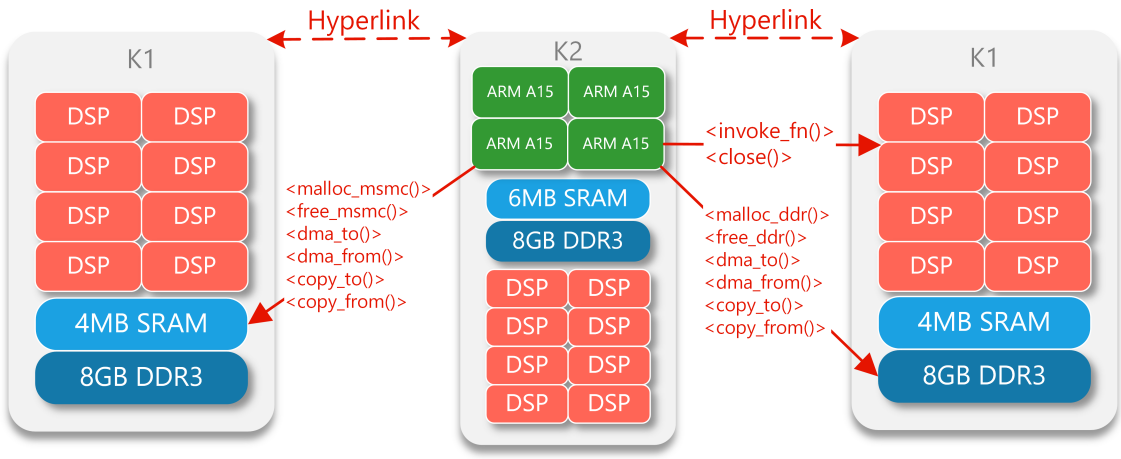


Figure 5.7: BrownDwarf Data transfer and work offload API

It then iterates in an infinite while loop. In each iteration it checks for incoming messages in the RX mailbox and executes the work item associated with that message. Once the work item is complete an ACK message is sent to the libk1comms on ARM to signal completion of the work request.

5.3 Building a hybrid fat binary for execution on BrownDwarf

The BrownDwarf system was designed to be scalable to hundreds of blades in multiple chassis configurations. Such configurations would have hundreds to thousands of nodes which would need to be usable via a HPC job scheduler such as SLURM or PBS. When such a scheduler launches a job across multiple nodes to be used with a distributed programming framework such as MPI, it is essential that the application binary execute in the same sequence across all nodes.

Since there are three separate LPSoCs in each node, three separate binaries require execution. The K2 *fat binary* execution across the ARM and DSP cores is orchestrated through use of the OpenMP 4.0 runtime environment as described in § 4.2. The K1 binaries can be individually loaded, started and stopped from K2 ARM cores using a proprietary loader library called `libdsploader.a` provided by Prodrive Technologies.

In order to successfully run an application across all three LPSoCs on a BrownDwarf node, orchestrating the correct execution sequence is critical. To

encapsulate this orchestration sequence across the three Keystone SoCs and have it performed automatically with no input required from the user, a new utility called *bdpack* was implemented. The objective of bdpack was to package up both the K2 and K1 binaries into a single hybrid fat binary along with driver code to orchestrate the required execution sequence of the individual binaries.

Figure 5.8 shows the process of creating the hybrid fat binary. A K2 binary and a K1 binary are the only two inputs required in this process. Once the K2 binary is compiled using all the required TI OpenCL, OpenMP, MPI and math library dependencies along with *libk1comms.so*; and the K1 binary is compiled and linked against *libk1dspmonitor.a* which already contains the bare-metal OpenMP runtime; they are fed into bdpack. Both the K2 and K1 binaries are converted into object files or *binary blobs* using the gcc linker. These object files are then linked into a driver program which is responsible for orchestrating the eventual execution. This link stage produces the hybrid fat binary.

This hybrid binary is also *self-extracting*. This implies that once it is run, it automatically extracts both the K2 and K1 binaries from within itself. This is possible since the driver program knows the start and end addresses of the K2 and K1 binary object blobs linked into the executable and uses this information to extract them from within.

Figure 5.9 shows the entire execution sequence of a hybrid fat binary across a BrownDwarf node. Once extraction is complete, `libdsploader` library operations are used to load and start executing the K1 binaries across Hyperlink. Following this, the hybrid binary forks itself to execute the K2 binary in one process and wait for all execution to complete in another. The K2 program then coordinates the K1 execution through use of the communication framework described in this chapter. Once the K2 completes execution, it joins the parent process. A final cleanup stage ensures that the K1 communication channels are properly closed and any extracted artifacts are not present.

5.4 Evaluation

It is essential to use all processing elements on a BrownDwarf node simultaneously to achieve maximum performance efficiency. Critical to this efficiency is the overhead of using the communication framework alongside other programming environments such as OpenMP 4.0 and MPI. This section describes a performance evaluation of a four-node BrownDwarf system using the matrix multiplication benchmark.

Matrix multiplication was implemented and evaluated on K2 SoCs in § 4.2.5.

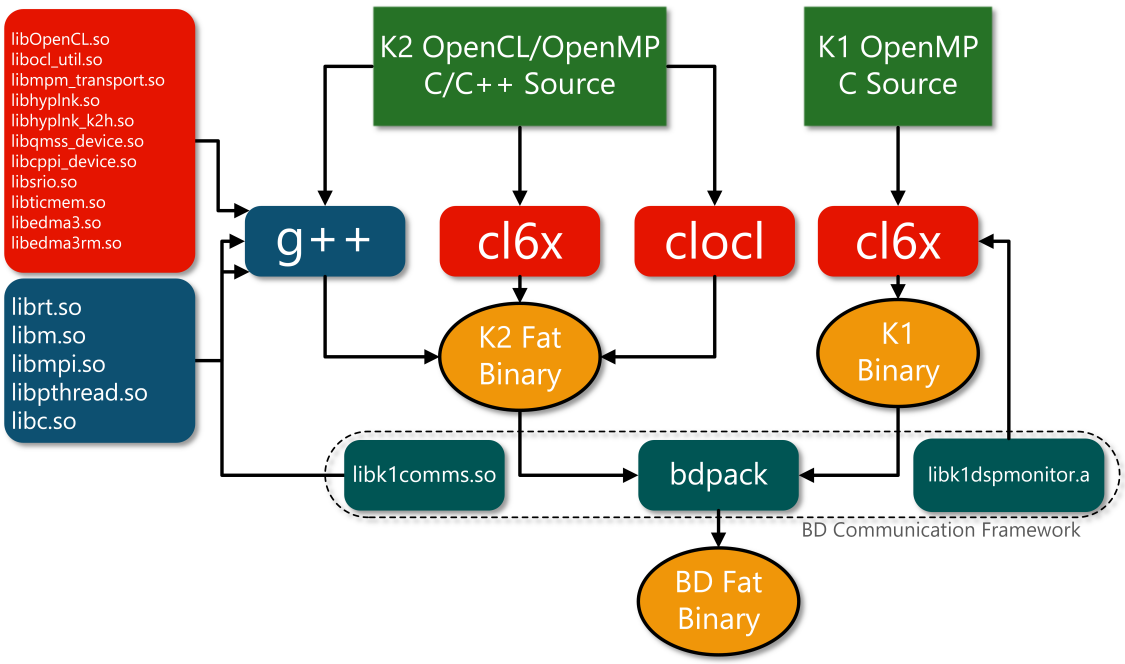


Figure 5.8: Build Sequence for Hybrid BrownDwarf Binary

It is logical to extend that evaluation to the BrownDwarf. The process of implementing matrix multiplication to use the communication framework on BrownDwarf is described in this section.

5.4.1 Partitioning GEMM: Adaptive search for best partition

With the overall objective of extracting the maximum amount of single and double-precision floating point performance for GEMM kernels across several different processing elements, it was essential to devise a scalable strategy to partition a given GEMM problem amongst the available devices.

To simplify the strategy, the two processing elements on K2, CPU and DSP were considered. Once a strategy was implemented, it could be extended to include more processing elements.

Two distinct experiments were performed. The first was to try and get the best performance from the CPU and DSP individually using optimized GEMM library implementations. The second experiment was to find a performance-optimal work-partition or split between the CPU and DSP in each system for a model problem size.

The BLIS library [Smith et al., 2014] was used on the ARM CPU cores on the K2. The best parallel performance over four cores for BLIS was achieved by

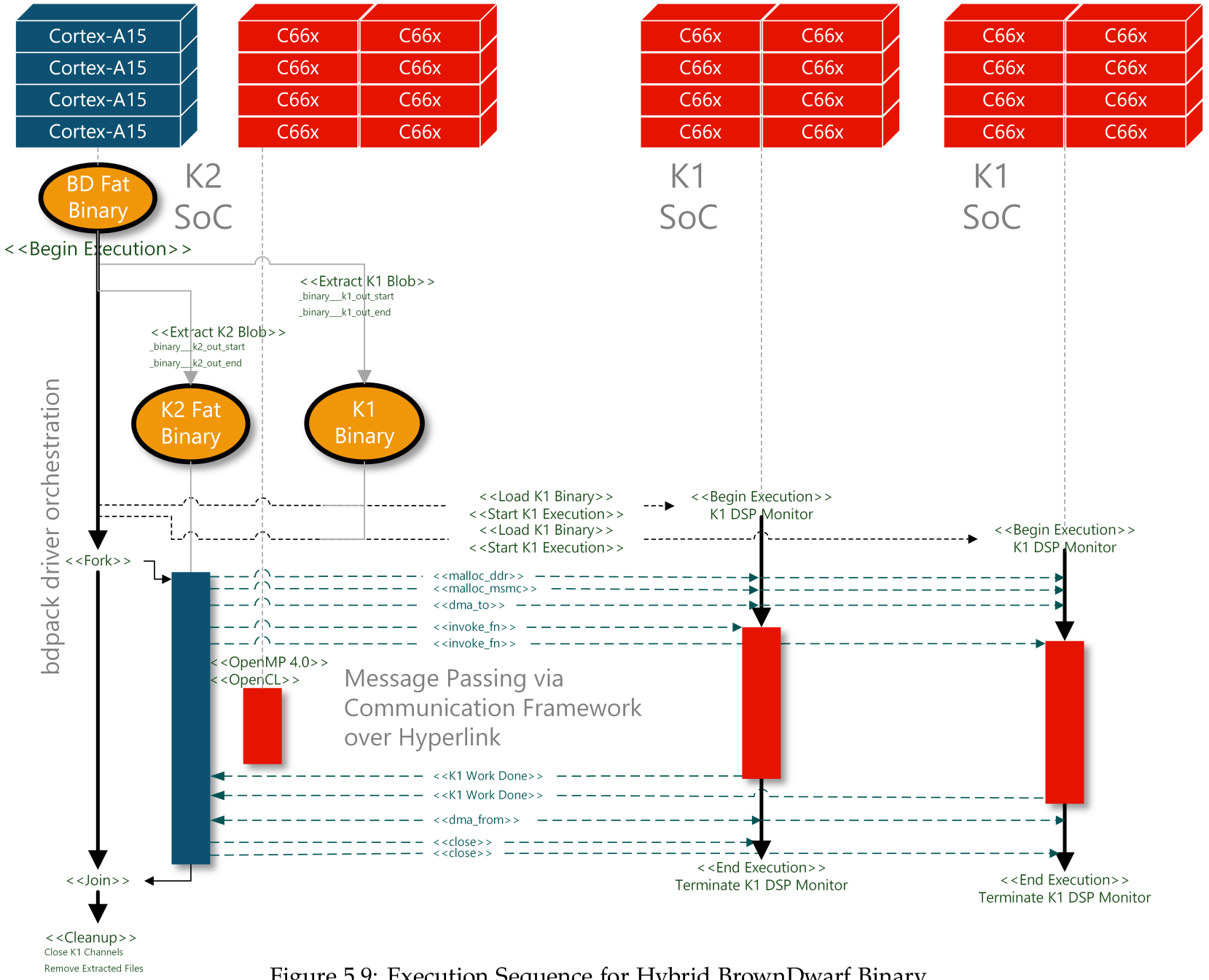


Figure 5.9: Execution Sequence for Hybrid BrownDwarf Binary

splitting the j_r loop into four parts (BLIS_JR_NT = 4). BLIS outperformed the auto-tuned ATLAS (version 3.11.37) library [Whaley and Dongarra, 1998] on the ARM Cortex-A15 and was therefore preferred.

Matrix multiplication can be easily subdivided into parts that can be computed in parallel. This enabled the second experiment to find a performance-optimal work partition between the CPU and DSP.

If input matrices are stored in column-major format, one natural way to partition the matrix product $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{C}, \mathbf{A}$ and $\mathbf{B}$ have dimensions $n \times m, n \times k$ and $k \times m$ respectively, is by columns in the manner

$$\left[\begin{array}{cccc} \mathbf{C_1} & \mathbf{C_2} & \ldots & \mathbf{C_p} \end{array} \right] = \mathbf{A} \times \left[\begin{array}{cccc} \mathbf{B_1} & \mathbf{B_2} & \ldots & \mathbf{B_p} \end{array} \right].$$

The product then consists of p independent matrix products $\mathbf{C_i} = \mathbf{A} \times \mathbf{B_i}$. The analogous approach of partitioning into rows, which may be beneficial for some input sizes or when matrices are stored in row-major form, is not considered here but should exhibit similar behavior. It is also important to note that another approach is possible where the matrix A is further subdivided into tiles and the algorithm is designed to multiply tiles of A with tiles of B. This is the approach adopted in the DSP SGEMM and DGEMM implementation and it has a lower memory footprint. However, for simplicity, the approach described above was adopted.

In order to find a performance-optimal split, one would presume that the measured peak performance of both CPU and DSP would be necessary. This was not the case since peak performance measured using BLIS on ARM and OpenMP 4.0 GEMM described in § 4.2.5 did not provide an empirical guide for determining the best split. This was primarily because the shape of the split matrices varied with the split and it was a major factor influencing performance.

As a result, an adaptive search algorithm was used to find the work-partition with the best performance without an expensive exhaustive search. This approach divides the work into only two pieces ($p = 2$); attempting to determine the optimal number of columns (p_1 and p_2) in each part prior to execution in order to split the work appropriately between the CPU and DSP.

- Arbitrarily set $p_1 = \lfloor \frac{m}{2} \rfloor$ and $p_2 = m - p_1$. The computation is then performed based on this partition and the CPU time, t_{CPU} and DSP time, (t_{DSP}) are measured.
- Get new values p_1' and p_2' by forcing $p_1' \frac{t_{\text{CPU}}}{p_1} = p_2' \frac{t_{\text{DSP}}}{p_2}$ (with $p_1' + p_2' = m$) and repeat until the values stabilize.
- Given p_1, p_2 found above, empirically test all partitions in the set $\{(p_1 -$

$\Delta pk, p_2 + \Delta pk) : k = -d, ...0, 1, ...d\}$ and pick the partition with best performance.

This approach is justified since performance does not necessarily vary smoothly as a function of the size. In practice the values $\Delta p = 16$ and $d = 2$ represent both fine-grained balancing and achieving a broad enough search.

This approach is particularly suitable in cases where a large number of multiplications of the same dimension have to be performed. In these cases, pretuning to determine the best split would only need to be performed once.

Considering specific implementation details on the K2, a single CPU thread is launched to issue work to the DSP asynchronously. Another thread calls the routine that performs the CPU portion of the computation, then blocks until the DSP kernel has finished execution. On the K2 system, shared memory accessed by both the CPU and DSP cores is allocated using the `__malloc_ddr` and `__malloc_msmc` API calls provided by the OpenMP 4.0 runtime environment.

5.4.2 Work partitioning across BrownDwarf nodes

Consider first work partitioning on a single BrownDwarf node. GEMM computation is partitioned using the adaptive algorithm described in § 5.4.1 between the K2 ARM, K2 DSP and 2×K1 cores. The BLIS GEMM implementation is used on the ARM cores. The OpenMP 4.0 implementation of GEMM described in § 4.2.2 is used on the K2 DSP cores. Using the message passing communication framework, GEMM is implemented for execution on the K1 DSP cores. This implementation is detailed in § C. It is used on both the K1 SoCs.

To compute $\mathbf{C}[\mathbf{M}, \mathbf{N}] = \mathbf{A}[\mathbf{M}, \mathbf{K}] \times \mathbf{B}[\mathbf{K}, \mathbf{N}]$, matrix A was replicated across all processing elements, while columns of B were partitioned such that each processing element computed a panel of output matrix C.

Input matrices were allocated in CMEM DDR memory shared between both ARM and K2 DSP cores as described in § 4.2.3. This enables both the ARM and K2 DSP cores to read and write to them simultaneously. These matrices were also stored in column major format since the DSP kernels were also implemented to accept column major input.

Based on given ratios of work distribution between the ARM cores, K2 DSP cores and K1 cores, a `node_work_distribution` matrix was populated which contained the number of columns (of matrix B) to be given to each processing element. Using this information, start column indices of matrix B for each processing element were calculated in `node_start_list`.

Figure 5.10 shows how the K1 computation was setup for each of the K1 SoCs. All of matrix A and panels of matrix B are copied to each K1 based on

the work partition. Upon completion of this setup, the computation is ready to begin across the node.

```
1 /* Setup K1 Compute */
2 cblas_dgemm_k1_setup(SOC_SHN0, A, B + node_start_list[K1_SHN0]*K, M,
3                       node_work_distribution[K1_SHN0], K, k1_threads);
4
5 cblas_dgemm_k1_setup(SOC_SHN1, A, B + node_start_list[K1_SHN1]*K, M,
6                       node_work_distribution[K1_SHN1], K, k1_threads);
```

Figure 5.10: K1 work setup

On the quad-core ARM processor, four threads are launched using an OpenMP parallel region to initiate DGEMM on each of the four individual processing elements. OpenMP tasks are used to launch the separate DGEMM kernels as shown in Figure 5.11. A `single nowait` clause was used to ensure that the four tasks are only launched by a single thread while the other threads hit the taskwait scheduling point and start to execute the launched tasks.

There are three separate calls to DGEMM kernels executing on different processing elements encapsulated in OpenMP tasks. The `cblas_dgemm()` function launches the ARM kernel across the four ARM cores. The `dsp_cblas_dgemm()` call encapsulates the OpenMP accelerator model implementation of DGEMM and is launched across the K2 DSP cores. The `cblas_dgemm_k1_compute()` function calls invoke the K1 DGEMM kernels.

Once the kernels complete, the results of the K2 ARM and DSP are already in K2 DDR memory. Results from the K1 SoCs are copied across to matrix C in K2 DDR memory using the `cblas_dgemm_k1_results()` function call as shown in Figure 5.12. At this point DGEMM computation is complete across all four processing elements on a single BrownDwarf node.

Distribution of work across multiple BrownDwarf nodes is performed using MPI. Similar to the work distribution principle used within a node, columns of matrix B are evenly provisioned across MPI nodes. Once the master node initializes the input matrices, they are broadcast across all nodes. The column provisioning is calculated and each node then further distributes their local provision amongst its four processing elements and computes their respective chunk of the result. Finally, results are collected in the master node using MPI send/receive operations.

5.4.3 Performance Analysis

The main objectives of the evaluation were to assess:

```
#pragma omp parallel default(shared) num_threads(4)
{
    #pragma omp single nowait
    {
        #pragma omp task
        {
            if (node_work_distribution[K2H_ARM] > 0)
            {
                cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                            M, node_work_distribution[K2H_ARM], K, alpha,
                            A,                                  /* lda = */ M,
                            B + node_start_list[K2H_ARM]*K,  /* ldb = */ K,
                            beta,
                            C + node_start_list[K2H_ARM]*M,  /* ldc = */ M
                            );
            }
        }

        #pragma omp task
        {
            if (node_work_distribution[K2H_DSP] > 0)
            {
                dsp_cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                                M, node_work_distribution[K2H_DSP], K, alpha,
                                A,                                  /* lda = */ M,
                                B + node_start_list[K2H_DSP]*K,  /* ldb = */ K,
                                beta,
                                C + node_start_list[K2H_DSP]*M,  /* ldc = */ M
                                );
            }
        }

        #pragma omp task
        {
            if (node_work_distribution[K1_SHN0] > 0)
            {
                cblas_dgemm_k1_compute(SOC_SHN0);
            }
        }

        #pragma omp task
        {
            if (node_work_distribution[K1_SHN1] > 0)
            {
                cblas_dgemm_k1_compute(SOC_SHN1);
            }
        }
    }
    #pragma omp taskwait
}
```

Figure 5.11: Work Distribution using OpenMP Tasks on ARM

```
1 /* Get Results from K1 */
2 cblas_dgemm_k1_results(SOC_SHN0, C + node_start_list[K1_SHN0]*M, M,
3                        node_work_distribution[K1_SHN0], &shn0_results);
4
5 cblas_dgemm_k1_results(SOC_SHN1, C + node_start_list[K1_SHN1]*M, M,
6                        node_work_distribution[K1_SHN1], &shn1_results);
```

Figure 5.12: K1 result retrieval

- performance of GEMM kernels running on BrownDwarf K1 SoCs
- performance overhead of the communications framework
- performance of GEMM kernels partitioned to run across all BrownDwarf processing elements
- performance of GEMM kernels running on multiple BrownDwarf nodes

A four node BrownDwarf cluster was used to conduct experiments for this evaluation. Each of the nodes was running on a Debian Linux filesystem using the Linux kernel 3.10.72. GCC 4.9.2 was used to compile code on the ARM, along with cl6x 8.1.2 for C66x DSP.

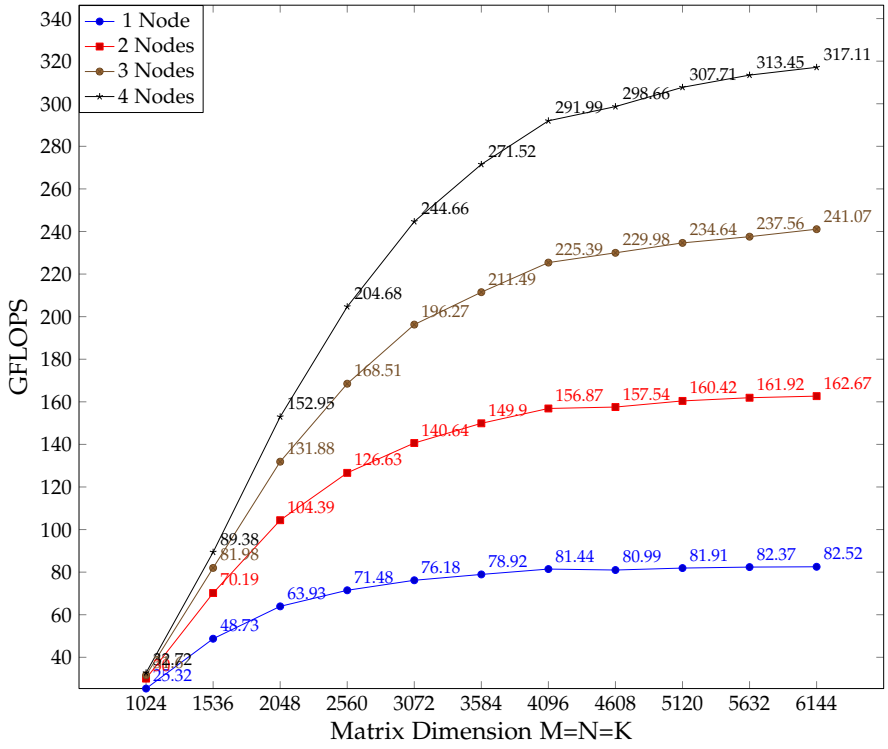
The TI OpenMP runtime version 2.03.01.00 was built into the K1 DSP Monitor and was used in bare-metal mode. Timing was performed on the ARM using the `MPI_Wtime()` timer provided by OpenMPI v2.0.1 being used on the BrownDwarf. The 10G ethernet link between the nodes is used for communication.

5.4.3.1 Using only K1 SoCs

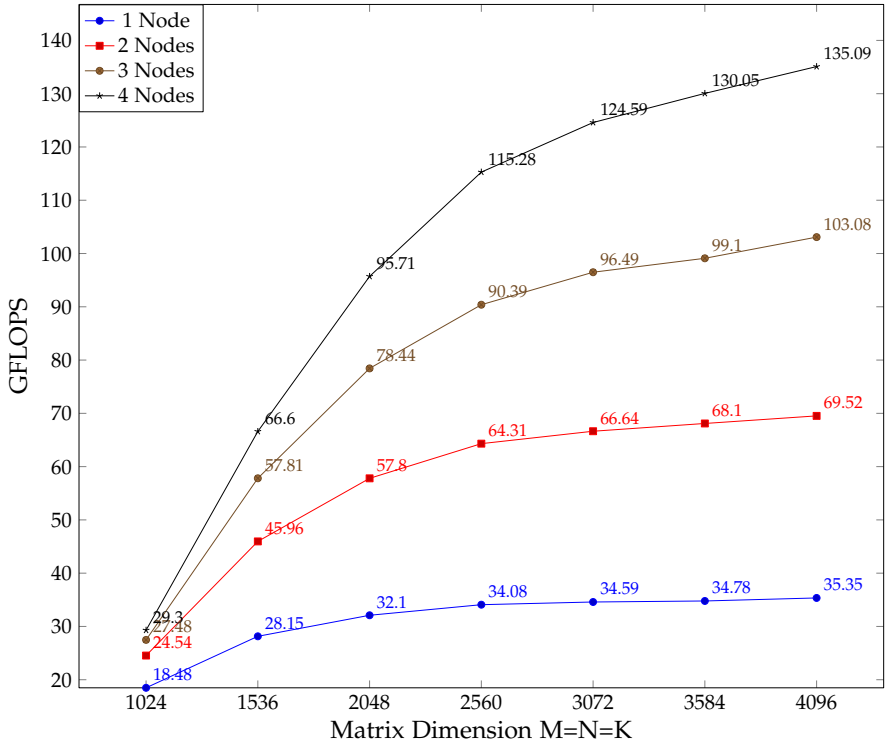
The objective of this experiment was to demonstrate scaling characteristics where problem size is increased along with number of K1 SoCs.

The performance of running GEMM solely on the K1 SoCs across multiple nodes is reported in Figure 5.13. In this experiment, GEMM was partitioned and computed across 2×K1 SoCs on each node. For example, in Figure 5.13(a), the smallest SGEMM experiment with 1024×1024 matrices first divides the computation across 2 K1 SoCs on 1 node. Then the computation is divided across 4 K1 SoCs on 2 nodes and eventually 8 K1 SoCs on 4 nodes.

The ARM cores and K2 DSP cores were not used to compute GEMM in this experiment. The computation is scaled from one to four BrownDwarf nodes on square input matrices with dimension sizes given in the x-axis. It is assumed that copies of input data are made available to each K1 before the computation



(a) SGEMM



(b) DGEMM

Figure 5.13: GEMM performance on two K1 SoCs

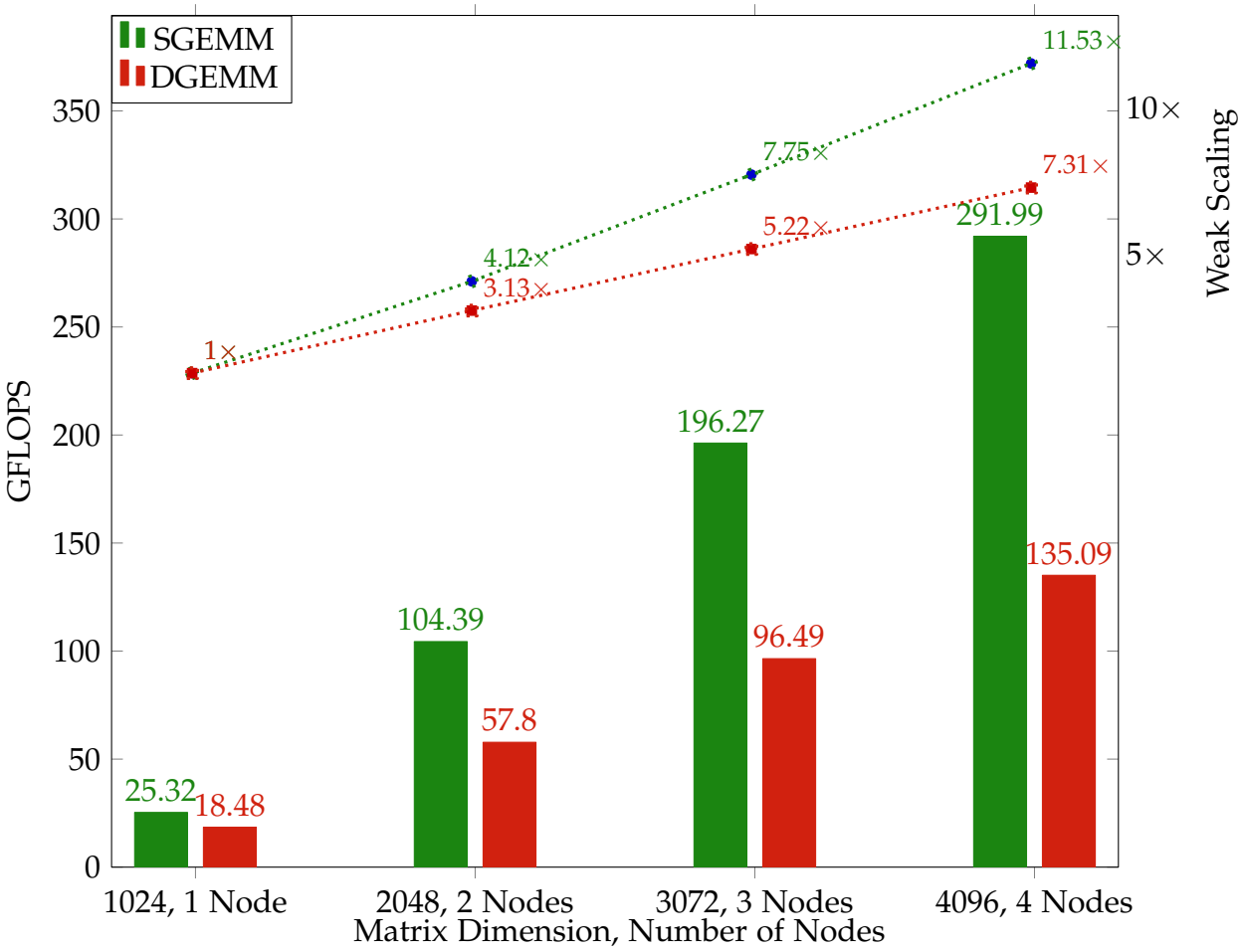


Figure 5.14: Weak Scaling Across BrownDwarf using only K1 DSP cores

begins and no data transactions are timed. That is, the reported performance does not include time taken to initialize input data on ARM DDR memory, transfer input data to K1 DDR memory or retrieve results from K1 DDR memory.

SGEMM performance is reported in Figure 5.13(a). On a single node, it is expected that computation on larger matrices will report higher performance until a peak value is reached signaling the performance limit of the application on specific processing elements. This is indeed the case as we observe the 1024×1024 matrix achieve 25.32 GFLOPS which scales up to 82.52 GFLOPS for the largest size of 6144×6144.

An individual K1 SoCs performance is therefore measured to be up to 41.26 GFLOPS. This is less than the measured performance of 74 GFLOPS SGEMM on a K1 SoC as reported in [Igual et al., 2012]. The reason for the drop in performance is due to only 171 KB of L2 SRAM being available on BrownDwarf K1 SoCs as compared to all 512 KB of L2 being used as SRAM. Since BrownDwarf supports any HPC application using OpenMP on the K1 SoC, this restriction of having only 171 KB SRAM available is mandatory given the rest of L2 is occupied by OpenMP runtime stack and L2 cache.

For each problem size, the measurements taken across multiple nodes represent strong scaling performance. For the largest problem size of 6144×6144, the performance scales from 82.52 GFLOPS on a single node to 162.67 GFLOPS on 2 nodes, i.e. a speed-up of $1.97\times$. Across 3 and 4-node experiments, the speed-ups achieved are $2.92\times$ and $3.84\times$ respectively. Near-linear speed-up is observed in this case.

DGEMM performance is reported in Figure 5.13(b). On a single node, for the smallest size of 1024×1024, performance of 18.48 GFLOPS is observed. Performance increases with problem size as expected and starts to plateau around the largest size of 4096×4096 with 35.35 GFLOPS. Similar to SGEMM, a reduction is observed compared to performance achieved by [Igual et al., 2012] due to less L2 SRAM being usable as scratch.

Strong scaling DGEMM performance for the largest problem size of 4096×4096 is also near-linear as observed for SGEMM. The speed-up factors of 2, 3 and 4-node experiments compared to a single node are $1.96\times$, $2.92\times$ and $3.82\times$ respectively.

Consider weak scaling performance of SGEMM and DGEMM in Figure 5.14. Speedups of $11.53\times$ and $7.31\times$ respectively are achieved when running on 4 nodes compared to a single node.

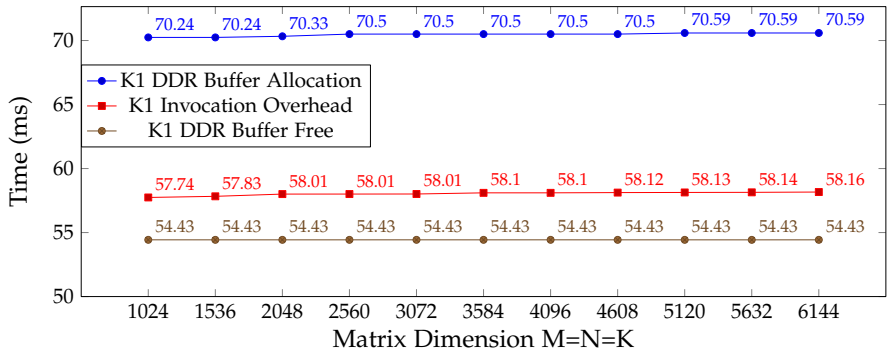


Figure 5.15: Overhead of Communication Framework - SGEMM

5.4.3.2 Communication Overhead

During this experiment, the overheads of the various communication operations were measured. This is in addition to time recorded and used for Figure 5.13. These overheads are reported in Figure 5.15. Three separate operations were profiled. These included the time taken to allocate a buffer in K1 DDR memory, the overhead of invoking a computation on a K1 SoC from K2 ARM cores and the time taken to free a buffer on K1 DDR memory.

The K1 DDR buffer allocation overhead measured time elapsed on the ARM cores to allocate K1 DDR buffers for all 3 matrices, A, B and C. Each such allocation operation included a message passed from the ARM to the K1 DSP monitor, the allocation being undertaken by the DSP monitor followed by a message passed back to the ARM for completion. This is observed to be constant irrespective of buffer size and measured to be around 70.5 ms. An individual allocation would therefore take in the order of 23.5 ms.

The overhead for invoking computation was measured by timing the computation both on the ARM and the K1 and taking the difference. It is also observed to be constant and measured to be around 58 ms.

Similar to buffer allocation, the K1 DDR buffer free operations were timed. This was also observed to be constant w.r.t buffer size and measured to be 54.43 ms. An individual free would therefore take around 18.14 ms.

5.4.3.3 Data Transfer Bandwidth

Measured DMA operation bandwidth is reported in Figure 5.16. The DMA transfers of input matrices A and B from ARM DDR to K1 DDR are profiled in this experiment. It is expected that higher data transfer sizes would saturate DMA channels across Hyperlink. This is indeed the case.

Consider the smallest problem size of 1024×1024. For SGEMM, the input

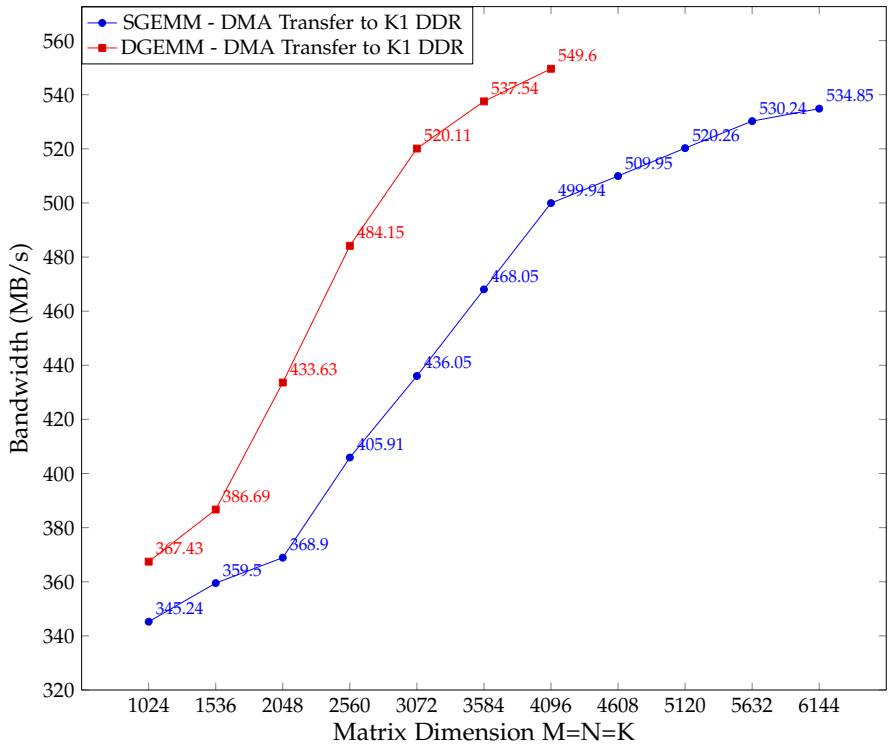


Figure 5.16: DMA Bandwidth

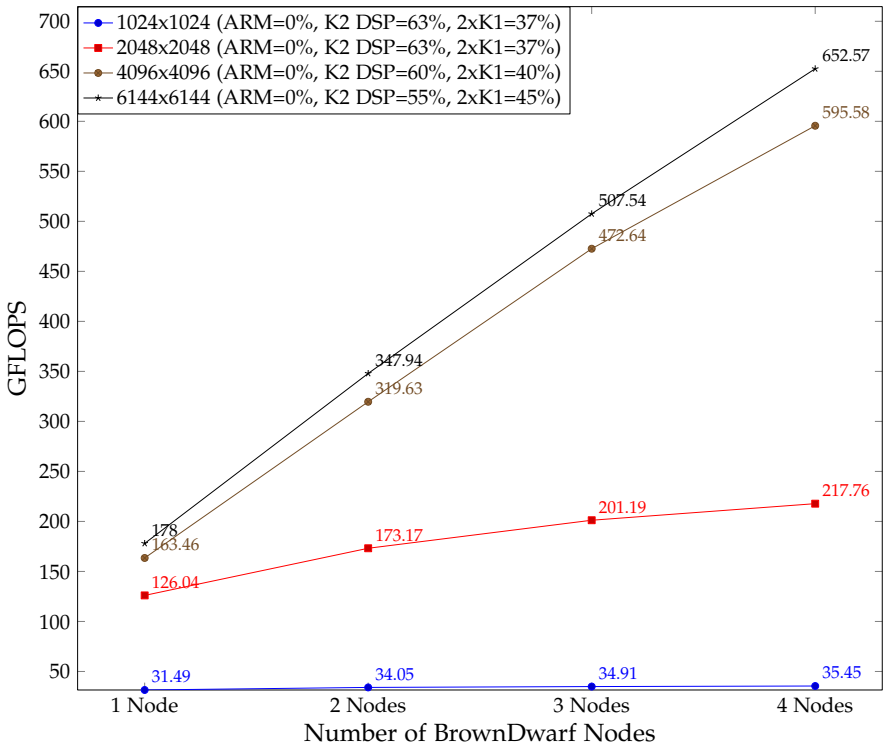
data size is $4MB$ for each of the two input matrices and 8MB in total. This transfer achieves 345.24 MB/s of bandwidth. For DGEMM, the input data size is $8MB$ each and 16MB in total. This transfer achieves 367.44 MB/s.

With increasing transfer sizes, the bandwidth is observed to increase linearly upto a transfer size of 128MB (4096×4096 SGEMM). For transfer sizes greater than 128MB, bandwidth starts to plateau. The highest measured bandwidth is 549.6 MB/s for the 256MB transfer size (4096×4096 DGEMM).

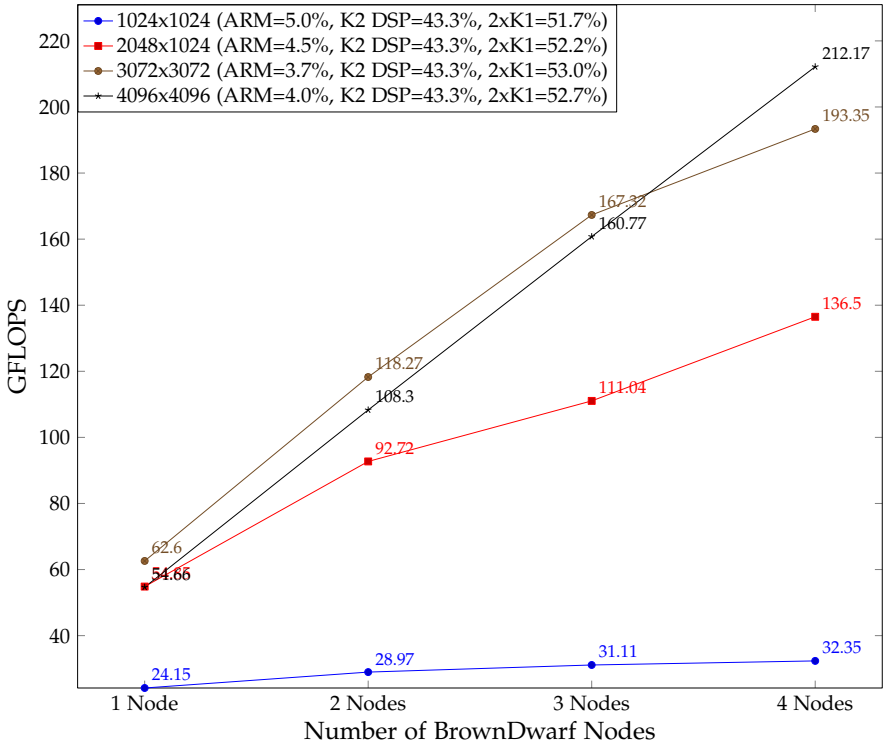
5.4.3.4 Using all processing elements

In order to effectively use all processing elements across a BrownDwarf system, the amount of work partitioned between different processing elements is critical to achieving good performance. To devise performance optimal work partitions within each node, the adaptive work partitioning algorithm described in § 5.4.1 was used.

The best performing partitions in the three-variable (ARM, K2 DSP, 2xK1) iteration space were chosen and their performance measured across multiple nodes. The partitions were made such that there is no overlap of work between them. These measurements are reported in Figure 5.17 and also indicate



(a) SGEMM



(b) DGEMM

Figure 5.17: GEMM strong scaling across BrownDwarf

strong scaling performance achieved across four nodes. In each case where a processing element was given work to do, the utilization of the processing element was maximized across all cores of the processing element. Each line in Figures 5.17(a)-5.17(b) represents a single workload size. For e.g. 1024×1024 represents multiplication of matrices of size 1024×1024.

Consider SGEMM performance reported in Figure 5.17(a). The best performing work partitions did not include any work being performed on the ARM cores. This clearly indicates how much more performant the DSP cores are compared to the ARM cores for SGEMM. Given that the K2 DSP cores had access to 768 KB of L2 SRAM, their performance was considerably better than K1 DSP cores with access to 171 KB SRAM. This is why K2 DSP cores were allocated larger work partitions. For a single node, the largest matrix size of 6144 × 6144 performs at 178 GFLOPS. Comparing this to the observed 82.52 GFLOPS only using K1 SoCs, performance increased by 2.12× by adding the K2 DSP cores. Across four nodes, performance increases to 652.57 GFLOPS with a speedup of 3.67× compared to a single node.

DGEMM performance is reported in Figure 5.17(b). In this case, the best performant partitions did include some work for the ARM cores, although in small quantities between 4 - 5%. The K2 DSP in this instance was allocated a fixed 43.3% of work while the ARM and K1 partitions were varied with problem size. For a single node, the largest matrix size of 6144 × 6144 performs at 54.66 GFLOPS. Comparing this to the observed 29.3 GFLOPS using only K1 SoCs, performance increased by 1.84× by adding the K2 DSP and ARM cores. Across four nodes, performance increases to 212.17 GFLOPS with a speedup of 3.88× compared to a single node.

5.5 Summary

This chapter presented work demonstrating the simultaneous use of both K2 and K1 SoCs for GEMM matrix multiplication through the implementation of a message passing communication framework for the nCore BrownDwarf System.

This is the first and only available implementation of a communication framework that allows programmatic use of the K1 SoCs with OpenMP code while being invoked from the ARM cores on a BrownDwarf node. Minimal and constant overheads of each communication operation were also observed.

Both SGEMM and DGEMM application codes were implemented for BrownDwarf node and significant performance was achieved on a four node BrownDwarf cluster for both application codes. Measured performance of 652.57 GFLOPS

for SGEMM and 212.17 GFLOPS of DGEMM were observed.

Bottlenecks in performance were identified in use of the Hyperlink interconnect. These represent significant limitations in the maximum read and write bandwidth available for HPC application codes and warrant further investigation in future work.

Overall, this work aggregates the OpenMP 4.0 runtime implementation described in chapter 4 along with a new communication framework to create a functional proof-of-concept programming environment that enables HPC application codes to execute across all processing elements on a BrownDwarf system.

Chapter 6

Energy efficiency and optimality on LPSoC processors

Chapter 3 considered optimizing performance of application codes on ARM CPUs present in LPSoC processors. Chapters 4 and 5 considered use of the accelerator within an LPSoC, partitioning work across both the CPU and the accelerator on an LPSoC and using multiple LPSoC processors to simultaneously work on a single problem. In this chapter, we look at the energy efficiency of LPSoC processors and the conditions for achieving energy optimality.

Key issues relating to the promise of high energy efficiency of LPSoC systems such as the TI Keystone II include i) absolute performance, ii) finding an energy-optimal balance in the use of the different on-chip devices and iii) understanding the performance-energy trade-offs when using the different on-chip devices.

Addressing the issue of absolute performance in LPSoCs and other heterogeneous systems is trivial. In the absence of parallel overhead, using CPU and accelerators simultaneously will lead to a *performance-optimal* work partition that maximizes absolute performance. In cases where the parallel overhead outweighs the performance advantage of using multiple devices, it is performance-optimal to use only the highest performant subset of devices.

Addressing the issue of energy consumption in LPSoCs and other heterogeneous systems is non-trivial. Different compute devices in heterogeneous systems have different energy and power consumption characteristics leading to a diverse set of possible choices of using these devices simultaneously to work on a single problem.

The amount of work distributed to each device, performance of individual devices for certain problems, the dynamic voltage frequency scaling (DVFS) configurations to use for certain devices, the active and idle power consumption of devices are some objective functions to consider in this *pareto analysis*. Pareto analysis is defined as a multi-objective optimization problem where more than

one objective function requires optimization simultaneously. A *pareto optimal* solution to such a problem is found when none of the objective functions can be improved in value without adversely affecting values of other objective functions. The goal of this analysis is to find a *pareto optimal* frontier to minimize energy consumption.

To simplify this pareto analysis, we reduce the objective functions in consideration by, i) disabling frequency scaling on devices and configuring them to operate at maximum frequency; ii) considering an embarrassingly parallel application code where communication overhead between devices would be negligible; iii) utilizing the best performant versions of the application for each device; and iv) considering only two devices, a host CPU and an accelerator. Under these conditions, a pareto frontier can be defined as a search for an *energy-optimal* work partition between two devices on a heterogeneous system.

It is not always evident what an *energy-optimal* work partition between a CPU and an accelerator might look like. Under the conditions of our analysis, a performance-optimal partition that minimizes idle time on both CPU and accelerator and minimizes *time-to-solution* for a problem may seem to be a likely candidate for an energy-optimal partition. Work presented in this chapter demonstrates that this is not always the case.

To explore this issue, a novel energy usage model is defined and validated. It is designed to predict the existence of an energy-optimal work partition between different processing elements on heterogeneous systems for any application. This model is validated by measuring performance and energy consumption of GEMM across five different heterogeneous systems.

Given the scope of our pareto analysis, this model is based on core system characteristics such as power consumption of devices in active and idle states along with maximum achieved performance rates of an application running individually on each processing element of a heterogeneous system.

It calculates the *energy-to-solution* and defines an optimality condition which can be used to predict a priori whether it is worth scheduling work on a particular processing element in order to save energy. Initial measurements are made to determine core system characteristics. The model is then used to make predictions based on these measured values.

In order to find a pareto frontier to minimize energy consumption and understand performance-energy trade-offs of using different processing elements, our first objective was to create a hardware and software framework to accurately measure energy usage during computation. This ability was lacking on several LPSoC systems due to absence of on-board current sensors. Work presented in this chapter outlines an environment for monitoring and responding

to energy usage.

Energy-to-solution measurements are made for GEMM kernels of different sizes running on five heterogeneous systems including the Keystone II using an adaptive work partitioning scheme. The results obtained are validated against predicted values from our energy usage model and it is observed that the model correctly predicts the existence of an energy-optimal work partition for all five systems.

The primary contributions of the work described in this chapter are:

- Design and implementation of a high resolution energy measurement framework for use with LPSoC platforms that enables measurement of energy as simply as measuring process time
- An energy usage model for heterogeneous systems which can be used to calculate and predict an energy-optimal work partition based solely on system characteristics and the computational rate of the application on each isolated device
- Validation of this model in the context of the important and widely used Level 3 BLAS operation of matrix multiplication on the Keystone II, two contemporary ARM based LPSoC platforms, the NVIDIA Tegra K1 and X1, and two Intel based conventional HPC systems containing Sandybridge and Haswell CPUs with NVIDIA K20 and K80 GPUs respectively
- Measurement of energy efficiency across these systems and a discussion of their performance-energy trade-offs

Section 6.1 describes the design of the novel high frequency energy measurement framework. Section 6.2 presents the energy usage model and its theoretical evaluation. Section 6.3 details the hardware features of the platforms used in experiments while section 6.4 explains the configuration of the experiments. Results are presented in section 6.5 along with detailed performance and energy efficiency analysis. Section 6.6 presents conclusion and future work.

6.1 Energy Measurement Framework for LPSoC systems

LPSoC devices rarely have internal energy measurement features. The objective was to create a high-resolution external energy measurement system to cater for such devices. To enable application codes to respond to energy usage during

runtime, another objective was to be able to start and stop energy measurement directly from the application code being executed on the LPSoC system similar to the measurement of process time.

The direct current (DC) used by LPSoC devices was measured using a high precision ammeter called the μCurrent Gold [Jones, 2010] and an mbed LPC1768 micro-controller with a 0-3.3V (12 bit) analog-to-digital converter (ADC).

The ADC is connected across the voltage output pins of the μCurrent Gold. Measurement start and stop signals are sent directly from within the application code running on the measured device via a serial link between the device and measuring computer as illustrated in Figure 6.1(a). A complete circuit diagram of this layout is presented in Figure 6.1(b).

Measurements are only taken while a computation is running, rather than trying to match timestamps or assuming changes in the power curve represent the start or end of the benchmark. Measurement data is sent via a serial link to an external computer at a rate of one reading every 10ms. These are numerically integrated using the midpoint rule to calculate the energy consumed by a computation.

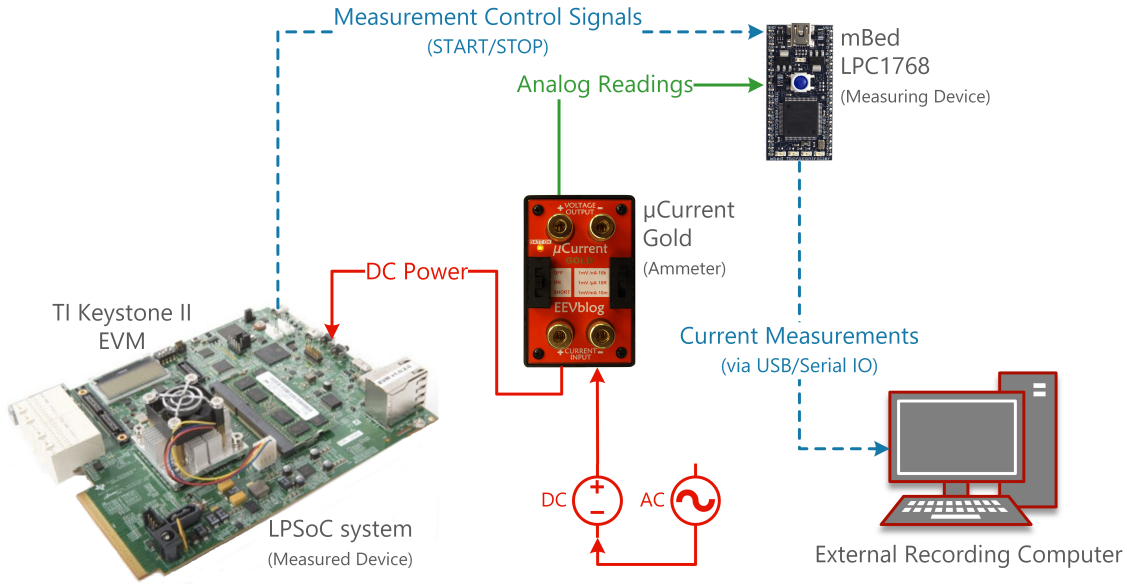
The μCurrent Gold is a current to voltage converter that uses a high precision, low value resistor and a precision two stage amplifier [Jones, 2010]. This results in negligible latency and accurate output. Used on the 1mV/mA setting, the precision of the μCurrent Gold is $\pm$0.1%, which is more precise than the ADC which has a resolution of 0.81$\pm$0.40mV. This corresponds to 0.81mA, which is 9.7$\pm$4.8mW at 12V.

An unmodified μCurrent Gold has a current measurement range of up to 1.25A. This was found to be insufficient for LPSoC systems being considered in this work. To extend the current measurement range of the μCurrent Gold, hardware modifications were made as shown in § 6.1.1. These modifications increased the range to the maximum supported by the ADC, i.e. 3.3A.

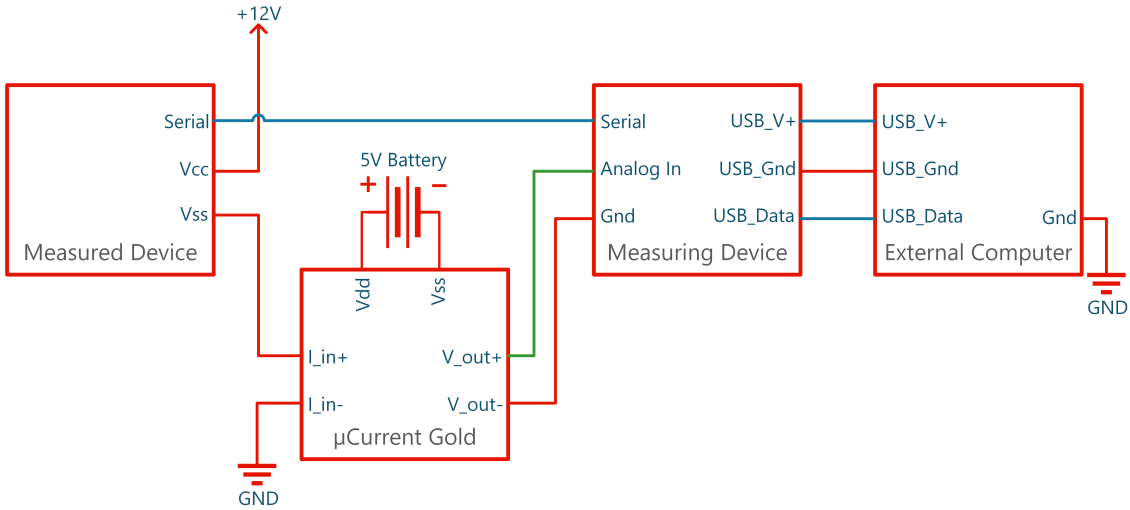
Measuring the DC from the power supply, rather than AC from the wall allows for a superior measurement. This is because AC varies over time with the grid frequency. Multiple AC measurements are required to create one value. Additionally, the frequency of the grid is much lower than the frequency of changes in an LPSoC system's power consumption. As a result, AC measurements can result in a loss of resolution. The AC to DC converter also introduces some overhead into the measurement. By measuring the DC power coming from the converter, this overhead is removed from our measurements.

The μCurrent Gold is relatively inexpensive at 70 USD and the mbed LPC 1768 micro-controller costs 55 USD. This brings the total cost of the energy measurement system to below 130 USD.

The energy measurement framework does not scale to multiple LPSoCs simultaneously. If that is required, the components of the framework would have to be replicated across each of the LPSoCs in considerations.



(a) Measurement Environment



(b) Circuit Schematic

Figure 6.1: LPSoC power measurement environment

6.1.1 Hardware modifications to the μCurrent Gold

The objective of hardware modifications was to increase the current measurement range of the μCurrent Gold as it was originally limited to 1.25A. An unmodified μCurrent Gold has a maximum output of 1.25V. From its schematics, it can be derived that this limit is a result of the power source. The original μCurrent source was a 3V battery.

This power source must supply both positive and negative lines due to the use of op-amps within the μCurrent. This was achieved within the μCurrent through the use of a floating 0V (or virtual ground), where the battery 0V serves as -V and the virtual ground is half-way between the battery 0V and +V. With a 3V battery, the maximum output is limited to 1.5V. The quoted maximum output is, however, at a lower level to account for voltage losses in the system.

For all experiments in this work, the 1mV/mA setting on the μCurrent was used. Since the μCurrent has a maximum output of 1.25V, the maximum measured current was limited to 1.25A. Initial measurements using a separate ammeter suggested that the Jetson TK1 platform would use up to 20W, or 1.67A at 12V under peak load. This maximum power usage was taken as a representative of contemporary LPSoC systems since other systems measured including the Keystone II had peak power consumption at a similar scale.

This prompted the need for the μCurrent's measurement range to be extended beyond 1.25A to suit LPSoC platforms such as the Jetson TK1. To allow a greater scale, the μCurrent's internal battery was replaced with a 5.0V (4.8V measured) battery pack. This provided a new scale of up to 2.4V (or 28.8W for TK1 since $2.4 \times 12 = 28.8$), allowing as much as 8.8W of headroom above the maximum expected power of 20W.

As only DC values above 0V were measured, a resistor was added to the μCurrent to lower the circuit virtual ground. Without alteration, the virtual ground is set to the center of the battery 0V and +V using a voltage divider with resistors of equal value.

A second resistor parallel to the resistor attached to the battery 0V and the center of the voltage divider was added. This effectively lowered the virtual ground, increased the positive range available to the measurement op-amps and decreased the negative range.

Since only positive values are measured, there is no detriment to the system with the decreased negative range. The positive range of measurement, however, was increased from 2.4V (2.4A in 1mV/mA mode) to approximately 4.0V (4.0A). This implies a new limit to the maximum voltage of the analogue input, resulting in a range of 0-3.3V (0-3.3A) since the ADC used within the μCurrent

could only convert up to 3.3V.

For LPSoC systems with DC current input rated at 12V, the power measurement range was now increased by 24.6W, going from the original 15W ($12 \times 1.25 = 15$) to the extended 39.6W ($12 \times 3.3 = 39.6$) using a modified μCurrent Gold. This extended range was adequate for measuring all LPSoC systems considered in this work.

6.2 Energy Usage Model

Consider two processing elements. For the purpose of the following, we take these to be a CPU and a GPU denoted by c and g respectively. These processing elements execute an application at the computational rates R_c and R_g.

When the application is active and executing, the power draw of the individual processing elements are P_c^a and P_g^a (W). When the processing elements are idle the power draws are P_c^i and P_g^i (W). The total computational cost of executing the application is labelled as N, with the fraction of the work given to the CPU denoted as f.

The times spent by the CPU and GPU executing the application are denoted as T_{ac} and T_{ag} (s) respectively. The total time to solution is labelled as T_s (s) where $T_s = max[T_{ac}, T_{ag}]$.

The total energy consumed is given in equation (6.2) where the first term in square brackets on the right-hand side represents the energy consumed by the CPU while the second term represents that consumed by the GPU. As $T_{ac} = \frac{fN}{R_c}$ and $T_{ag} = \frac{(1-f)N}{R_g}$, equation (6.2) can be rearranged in the form $E(f) = N \max\left(m_1 f + c_1, m_2 f + c_2\right)$ where m_1, c_1, m_2, c_2 are constants. $E(f)$ is therefore an upper envelope of two straight lines.

$$
\begin{aligned}
E(f) &= E_c(f) + E_g(f) \\
E(f) &= \left[(P_c^a - P_c^i)\frac{Nf}{R_c} + P_c^i T_s\right] + \left[(P_g^a - P_g^i)\frac{N(1-f)}{R_g} + P_g^i T_s\right] && (6.1) \\
&= N\left(\frac{(P_c^a - P_c^i)f}{R_c} + \frac{(P_g^a - P_g^i)(1-f)}{R_g}\right) + (P_c^i + P_g^i)T_s \quad \text{[J]} && (6.2)
\end{aligned}
$$

where,

$$
\begin{aligned}
T_{ac} &= \frac{fn}{R_c} \quad \text{[s]} && (6.3) \\
T_{ag} &= \frac{(1-f)N}{R_g} \quad \text{[s]} && (6.4) \\
T_s &= max[T_{ac}, T_{ag}] \quad \text{[s]} && (6.5)
\end{aligned}
$$

Energy consumed by CPU and GPU are as follows,

$$\begin{aligned} E_c(f) &= P_c^a T_{ac} + P_c^i (T_s - T_{ac}) \\ &= P_c^a \frac{nf}{R_c} + P_c^i (T_s - \frac{nf}{R_c}) \\ &= (P_c^a - P_c^i)\frac{nf}{R_c} + P_c^i T_s \qquad \text{[J]} \quad (6.6) \\ E_g(f) &= P_g^a T_{ag} + P_g^i (T_s - T_{ag}) \\ &= P_g^a \frac{N(1-f)}{R_g} + P_g^i (T_s - \frac{N(1-f)}{R_g}) \\ &= (P_g^a - P_g^i)\frac{N(1-f)}{R_g} + P_g^i T_s \qquad \text{[J]} \quad (6.7) \end{aligned}$$

The energy-optimal work partition is the value of f that minimizes equation (6.2) in the interval $f \in [0,1]$. Its minimum must either occur at $f = 0$, $f = 1$ or where the two lines $(m_1 f + c_1$ & $m_2 f + c_2)$ intersect. We label the point of intersection as $f = f^*$. This occurs when $T_{ac} = T_{ag}$, i.e. when the CPU and GPU finish exactly together with no idle time. Therefore, f^* is given by,

$$\frac{Nf*}{R_c} = \frac{N(1-f*)}{R_g} \Rightarrow f^* = \frac{R_c}{R_g + R_c} \Rightarrow f^* R_g = (1-f^*)R_c \Rightarrow f^*(R_g + R_c) = R_c \quad (6.8)$$

To derive the conditions under which f^* represents an energy-minimum we substitute the values $f = 0$, $f = 1$ and f^* into equation (6.2) as follows,

$$E(0) = N\frac{P_g^a - P_g^i}{R_g} + (P_c^i + P_g^i)N\frac{1}{R_g} \qquad = N\left(\frac{P_g^a + P_c^i}{R_g}\right) \quad (6.9)$$

$$E(1) = N\frac{P_c^a - P_c^i}{R_c} + (P_c^i + P_g^i)N\frac{1}{R_c} \qquad = N\left(\frac{P_g^i + P_c^a}{R_c}\right) \quad (6.10)$$

$$E(f^*) = N\left(\frac{P_c^a + P_g^a}{R_g + R_c}\right) \quad (6.11)$$

$$\begin{aligned} E(f^*) &= N\left(\frac{(P_c^a - P_c^i)f^*}{R_c} + \frac{(P_g^a - P_g^i)(1-f^*)}{R_g}\right) + (P_c^i + P_g^i)\max\left(\frac{nf^*}{R_c}, \frac{N(1-f^*)}{R_g}\right) \\ &= N\left(\frac{(P_c^a - P_c^i)}{R_c}\frac{R_c}{R_g + R_c} + \frac{(P_g^a - P_g^i)}{R_g}\frac{R_g}{R_g + R_c}\right) \\ &+ N(P_c^i + P_g^i)\max\left(\frac{1}{R_c}\frac{R_c}{R_g + R_c}, \frac{1}{R_g}\frac{R_g}{R_g + R_c}\right) \\ &= N\left(\frac{P_c^a - P_c^i + P_g^a - P_g^i}{R_g + R_c} + \frac{P_c^i + P_g^i}{R_g + R_c}\right) \quad (6.12) \end{aligned}$$

For f^* to represent the energy-minimum, the following conditions would have to hold, $E(f^*) < E(0)$ and $E(f^*) < E(1)$. Using equations (6.9), (6.10) and (6.12),

we obtain the following inequalities,

$$\frac{P_c^a + P_g^a}{R_g + R_c} < \frac{P_g^a + P_c^i}{R_g} \tag{6.13}$$

$$\frac{P_c^a + P_g^a}{R_g + R_c} < \frac{P_g^i + P_c^a}{R_c} \tag{6.14}$$

Rearranging (6.13) and (6.14) gives the following,

$$\frac{P_c^a - P_c^i}{P_g^a + P_c^i} < \frac{R_c}{R_g} < \frac{P_c^a + P_g^i}{P_g^a - P_g^i} \tag{6.15}$$

Equation (6.15) indicates that an energy-optimal execution scheme that uses both the CPU and the GPU will exist depending on the ratio of execution rate between the CPU and GPU and how this compares with two quantities related to active and idle power usage of these devices. It does not depend on problem size N.

6.2.1 Theoretical Evaluation

The objective of this theoretical evaluation of the energy usage model is to try and identify the conditions under which the model might predict an energy-optimal work partition. This evaluation also helps in deeper understanding of the model and its implications.

Consider a system where the performance of both the CPU and GPU is 100 GFLOPS, the idle power of each is set to 3W and the active power usage of each varies from 0-18 W. Let the application be the multiplication of two square matrices of dimension 4096 that is partitioned by columns between the CPU and the GPU.

Figure 6.2 shows the total energy used for a variety of active power states for the CPU and the GPU. In all cases, there exists an energy minimum when both the CPU and the GPU are utilized.

In Figure 6.3, the GPU active power is set to 10W while the CPU active power is varied. Under this scenario, each curve has an inflection point at the halfway mark of 2048 columns (since the performance of CPU and GPU are equal) but the slope of the first half of each curve is not always negative. Around $P_c^a = 15W$, the slope of the first half turns positive implying that the energy minima would only occur when no work is given to the CPU. This is also verifiable from equation 6.15. We see that for $P_c^a \in [5, 15]$, the inequality holds i.e. $[0.15, 0.92] < 1 < [1.14, 2.57]$. However, for $P_c^a = 18W$, it evaluates to $1.15 < 1 < 3$ which does not hold and there is no minimum.

Other scenarios can be explored. For example, if P_g^a was varied while keeping other factors constant, we would obtain a mirror image of Figure 6.3. Similarly, if the performance of the CPU were lowered while keeping other factors constant, the energy-minimum would shift towards allocating more work to the GPU and eventually to $x = 0$ implying that it is no longer energy-optimal to allocate any work to the CPU.

Using 6.15 it is now possible to estimate whether an energy-optimal balance exists for this system. Substituting the values into the lower and upper bounds, we get $\frac{P_c^a - P_c^i}{P_g^a + P_c^i} = \frac{[3,10]-3}{[3,10]+3} = \frac{[0,7]}{[6,13]} = [0, 0.54]$ and $\frac{P_c^a + P_g^i}{P_g^a - P_g^i} = \frac{[3,10]+3}{(3,10]-3} = \frac{[6,13]}{(0,7]} = [1.3, 7]$. Also, $\frac{R_c}{R_g} = \frac{100}{100} = 1$. Therefore, the inequality holds i.e. $[0, 0.54] < 1 < [1.3, 7]$ and an energy-minima always exists when both CPU and GPU cores are used in tandem.

In fact, it is evident from the graph, that the energy-optimal balance in this case occurs when both CPU and GPU have equal work of 2048 columns. Note that the performance and active power are set to be equal in this case for both CPU and GPU, which leads to a balance between energy and performance. The model also exposes an issue in our active power state of 3W when the divisor of the upper bound, $P_g^a - P_g^i = 0$. This renders the upper bound undefined in this instance and explains an inherent limitation that P_g^a and P_g^i may not be equal i.e. CPU or GPU active power may not be equal to its idle power. This is unlikely to occur in practice.

It is also evident that the total energy consumption is directly proportional to the active power. Therefore, if the CPU and GPU have equivalent active, idle power states and equivalent performance, it is always beneficial from an energy perspective to partition work equally across both the CPU and GPU. Further, intuitively it can be deduced that if the active power states and performance were fixed, the system would consume minimal energy when both the CPU and GPU are never idle.

Figure 6.5 demonstrates the scenario of varying idle power of both the CPU and GPU cores between 0W and 10W. We assume that the idle power should never exceed active power and therefore fix the active power at 10W. When the system consumes 0W at idle, the energy consumption is uniform regardless of how much work is partitioned across the CPU or GPU. At any other value of idle power consumption, the system always has the same energy minima which is equal to energy consumption when idle power is 0W.

Equation 6.15, however, can only be used for idle power states of 2, 4, 6, and 8W and it holds for these states i.e. $[0.11, 0.67] < 1 < [1.5, 9]$ implying the existence of the energy-minima. This minima therefore corresponds to the state

when both the CPU and GPU are never idle during computation i.e. when the work partition is perfectly balanced or $T_{ac} = T_{ag}$. If the performance values were varied however, this minima would shift. This implies that when the active power states and performance are equivalent, the system would consume minimal energy when both the CPU and GPU are never idle.

Finally, varying the performance of either the CPU or GPU cores provides more insight. Figure 6.6 fixes active power at 10W and idle power at 3W for both CPU and GPU. The performance of the GPU is fixed at 100 GFLOPS while that of the CPU ranges between 30 GFLOPS and 100 GFLOPS.

An energy minima does not always exist under these conditions. More specifically, as the CPU performance reduces, the minima shifts towards allocating more work to the GPU. Using equation 6.15 we find that the inequality holds for $0.54 < [0.6, 1] < 1.85$ which corresponds to $R_c \in [60, 100]$GFLOPS and does not hold for lower values i.e. $R_c \in [0, 54]$GFLOPS.

This can be clearly observed from the curve for CPU = 50 GFLOPS and lower where the slope changes from negative to positive for the first half of the curve and the minima shifts to $x = 0$ which implies it is no longer energy-optimal to allocate work to the CPU.

6.2.2 Adaptation to LPSoC platforms

Often low-power SoC platforms are single-board computers that do not allow for individual measurement of CPU and GPU power usage; for such systems it is only possible to measure power at the board level. Our energy measurement framework described in § 6.1 enables this. Under this scenario, it is useful to recast equation (6.15) in terms of measurable states.

We denote the measurable state when both the CPU and the GPU are active as P_{acg} (W), and that when both are idle as P_{icg} (W). The state when the CPU is active and GPU idle is denoted as P_{acig} (W) while that when CPU is idle and GPU is active is denoted as P_{icag} (W). Relations between these measurable states and individual power states can be empirically derived:

$$P_{acg} = P_c^a + P_g^a; P_{icg} = P_c^i + P_g^i; \tag{6.16}$$

$$P_{acig} = P_c^a + P_g^i; P_{icag} = P_c^i + P_g^a \tag{6.17}$$

$$P_c^a - P_c^i = P_{acig} - P_{icg} \tag{6.18}$$

$$P_g^a - P_g^i = P_{icag} - P_{icg} \tag{6.19}$$

Using (6.17), relations between measurable system states and individual com-

ponent power states can be derived,

$$P_c^a - P_c^i = P_{acig} - P_{icg} \tag{6.20}$$

$$P_g^a - P_g^i = P_{icag} - P_{icg} \tag{6.21}$$

Substituting (6.20), (6.21), (6.17) in (6.2),

$$E(f) = N[\frac{(P_{acig} - P_{icg})f}{R_c} + \frac{(P_{acg} - P_{acig})(1-f)}{R_g}] + P_{icg}T_s \qquad \text{[J]} \tag{6.22}$$

$$= N\left(\frac{(P_{acig} - P_{icg})f}{R_c} + \frac{(P_{icag} - P_{icg})(1-f)}{R_g}\right) + P_{icg}T_s \qquad \text{[J]} \tag{6.23}$$

Equation (6.21) has an equivalent form, $P_g^a - P_g^i = P_{acg} - P_{acig}$, and can be used for substitution into (6.23) based on which constant values between P_{acg} or P_{icag} are more accurately measured. Equation (6.23) can be used to model the energy consumption of a system with work divided between the CPU and GPU processors once the constants $P_{acig}, P_{icg}, P_{acg}, R_c, R_g$ are determined.

Using (6.20) and (6.21) with our energy-optimality condition given in equation (6.15), the equivalent condition in terms of measurable power states of LP-SoC systems is as follows,

$$\frac{P_{acig} - P_{icg}}{P_{icag}} < \frac{R_c}{R_g} < \frac{P_{acig}}{P_{icag} - P_{icg}} \tag{6.24}$$

6.2.3 Critique of Energy Usage Model

We note that while the description above has been for two devices which we call the CPU and the GPU, the model is entirely general and could be extended to larger systems with multiple devices since both power and performance characteristics are additive. This would also apply to multiple nodes since their CPU and accelerator components can be considered individually.

The model also points to critical processor design characteristics such as the ratio between active and idle power and how this fundamentally influences the way heterogeneous devices should be programmed if energy usage is to be minimized.

As we measure energy for SoC boards system-wide, it includes CPU, memory and other components. For conventional heterogeneous systems with attached accelerators our model applies immediately when the input data is already in place (replicated or partitioned). This is applicable even if memory copies are overlapped with computation. The model also provides an idealized result. A separate memory copy would effectively reduce the FLOP rate and

could be modelled as such. For SoC systems with shared physical memory there are no data transfers between compute devices and therefore, the model applies immediately.

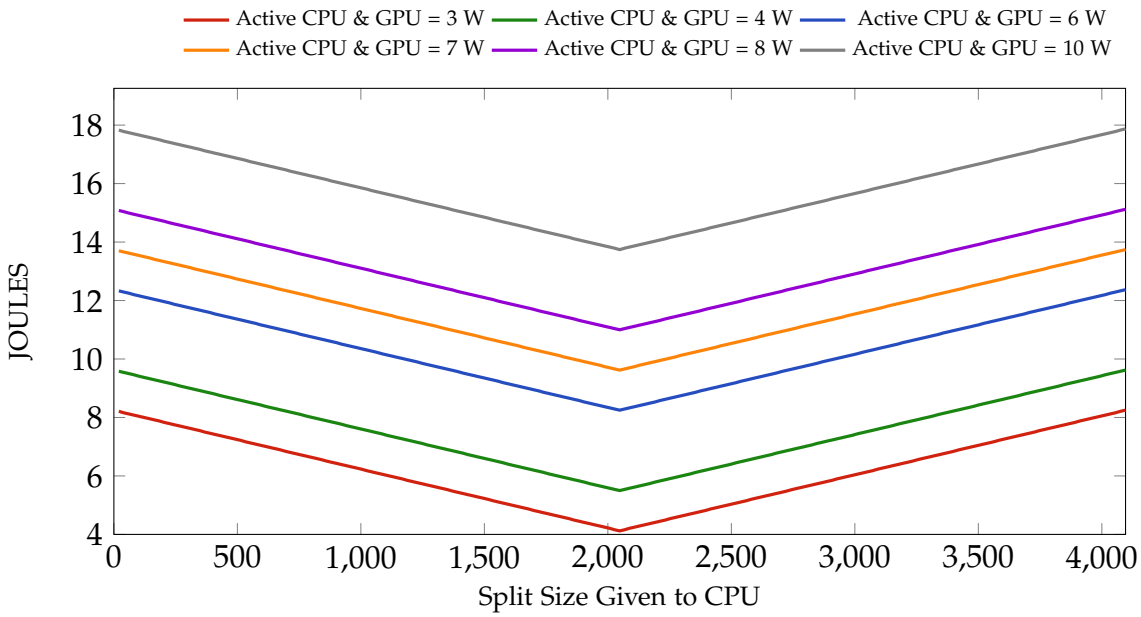


Figure 6.2: Energy Usage Model: Active (CPU & GPU = ?); Idle (CPU & GPU = 3W); GFLOPS (CPU & GPU = 100)

6.3 Hardware platforms

The three LPSoC platforms and two Intel-based platforms used in our experiments are described in table 6.1.

For the LPSoC platforms, the Keystone II Evaluation Module Rev 4.0 was used. It is denoted as *K2*. Our implementation of OpenMP 4.0 runtime environment described in § 4 was used to offload computation to the K2 DSP cores.

Two contemporary NVIDIA LPSoC platforms released post 2014 were used. The Jetson TK1 development kit (*TK1*) contains an NVIDIA Tegra K1 SoC with a quad-core ARM Cortex-A15. The Jetson TX1 development kit (*TX1*) has an NVIDIA Tegra X1 SoC which houses a 64-bit quad-core ARM Cortex-A57. CUDA was used offload kernels to the GPU cores.

On all three of these LPSoC systems, CPU and accelerator cores were set to either maximum or minimum frequency depending on whether they were being used (active) or were idle respectively during an experiment.

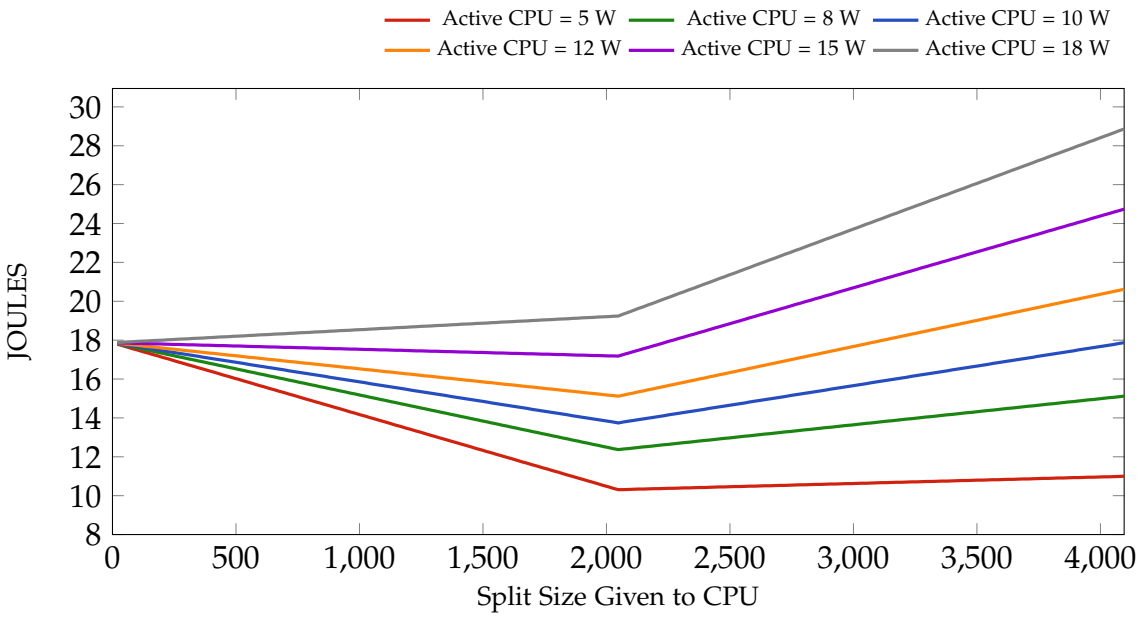


Figure 6.3: Energy Usage Model: Active (CPU = ?, GPU = 10W); Idle (CPU & GPU = 3W); GFLOPS (CPU & GPU = 100)

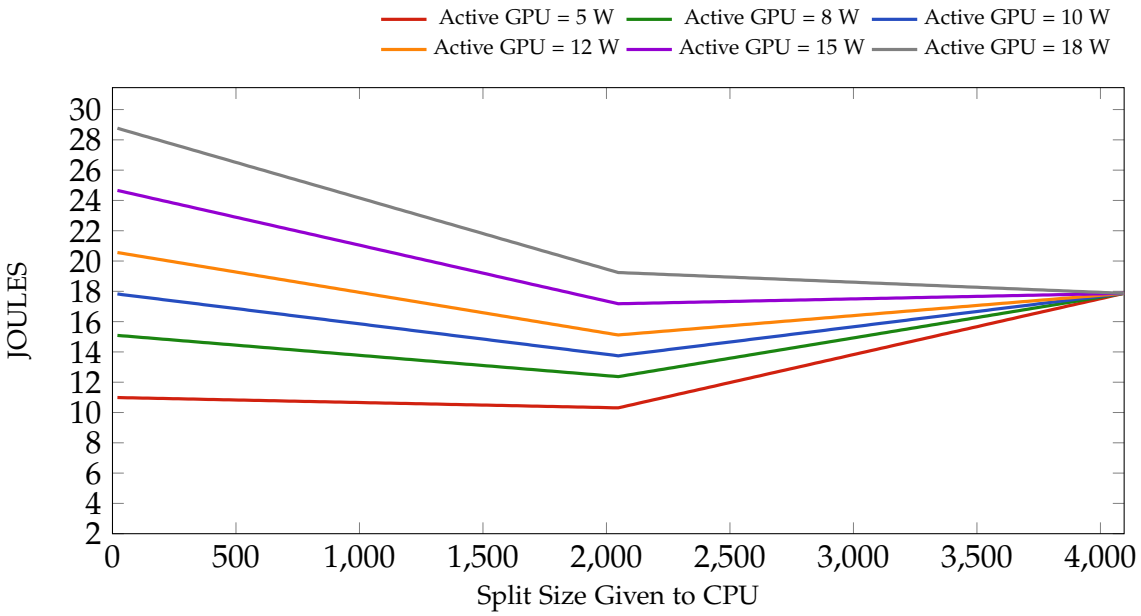


Figure 6.4: Energy Usage Model: Active (CPU = 10W, GPU = ?); Idle (CPU & GPU = 3W); GFLOPS (CPU & GPU = 100)

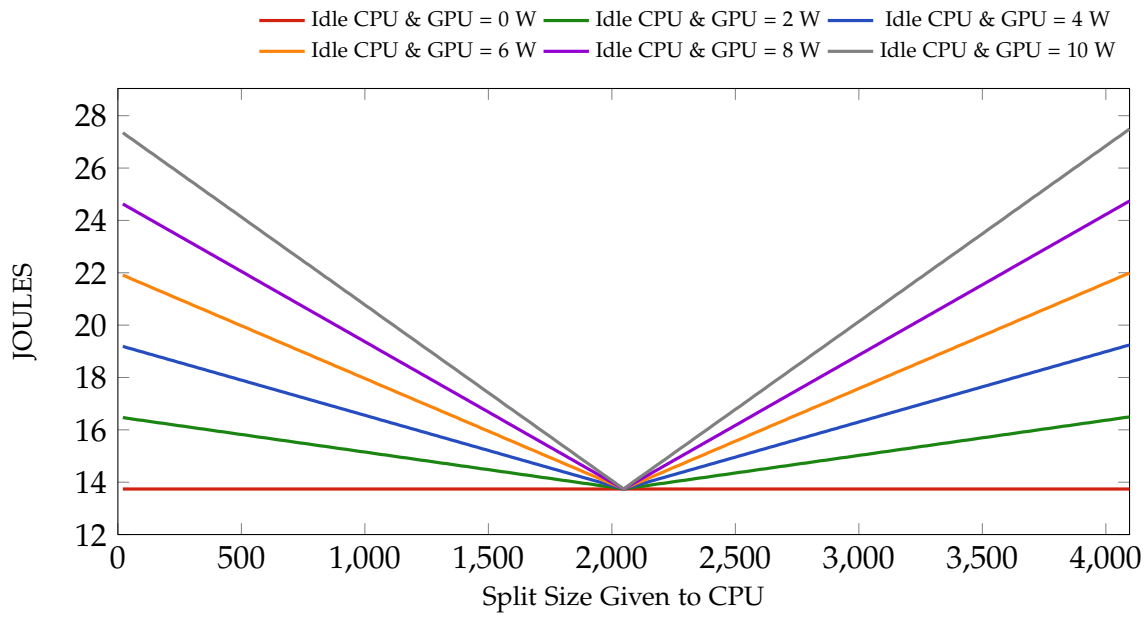


Figure 6.5: Energy Usage Model: Active (CPU & GPU = 10W); Idle (CPU & GPU = ?); GFLOPS (CPU & GPU = 100)

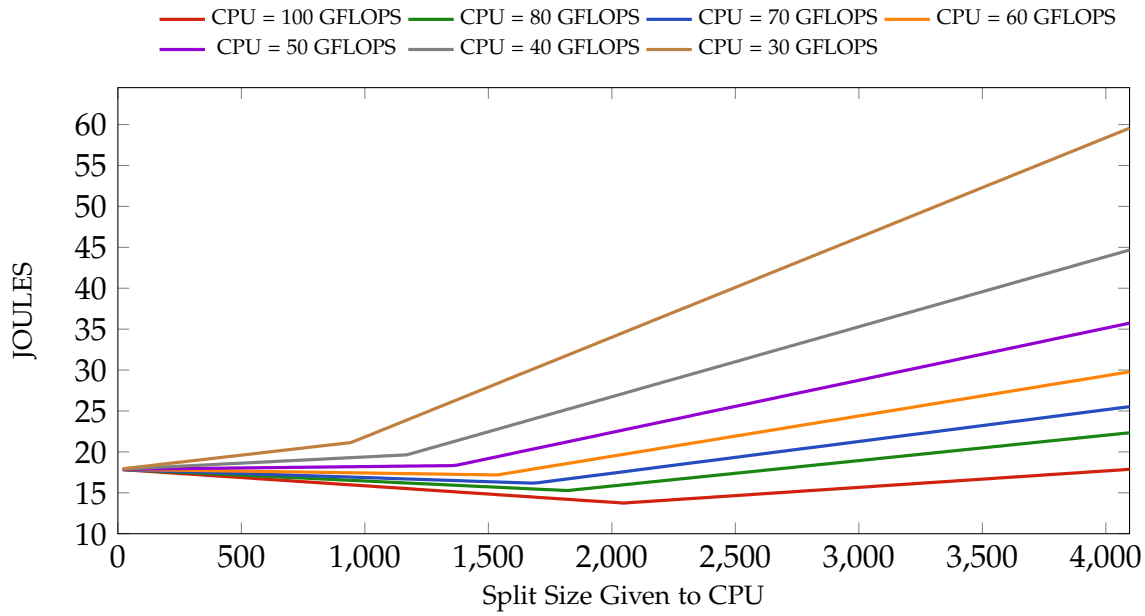


Figure 6.6: Energy Usage Model: Active (CPU & GPU = 10W); Idle (CPU & GPU = 3W); GFLOPS (CPU = ?, GPU = 100)

For the conventional Intel-based HPC systems with attached accelerators, the *SANDY* system contains dual-socket Xeon E5-2665 processors and a discrete NVIDIA Tesla K20m card. The *HASWELL* system contains dual-socket Xeon E5-2620 v3 Haswell processors and a discrete NVIDIA Tesla K80 card which houses two GK210 GPUs.

As the two GK210 GPUs in the K80 are effectively separate devices with physically separate memory (and CUDA identifies them as such) we use just one GPU for our experiments for fair comparison. Power measurements are also reported for one GPU on this system.

6.4 Experimental Setup

Adaptive work partitioning as described in § 5.4.1 is used to distribute GEMM across CPU and accelerator on each system. To our knowledge, the best SGEMM and DGEMM implementations available for each host CPU was used.

On the TK1 and TX1 and K2 systems with ARM hosts, the BLIS library [Smith et al., 2014] was used on the ARM cores with the appropriate configuration (e.g. Cortex-A15 on TK1) as described in § 5.4.1.

The possibility of using the Intel MKL library was also considered on the Intel based systems. On SANDY, it gave comparable performance to auto-tuned ATLAS (version 3.11.37). On HASWELL, good scaling could not be achieved over two sockets with MKL even with recommended NUMA policies. As ATLAS achieved close to peak theoretical performance, ATLAS was used on the Intel systems.

GEMM was programmed on the GPU cards using the CUDA programming language. Implementation of GEMM provided in the vendor-supplied cuBLAS library was used.

On the Keystone II, the GEMM implementation described in § 4.2.5 using the OpenMP 4.0 runtime environment was used to program the DSP cores. Shared memory accessed by both the CPU and DSP cores on the K2 was allocated using the `__malloc_ddr` and `__malloc_msmc` API calls provided by the OpenMP 4.0 runtime environment.

On both the TK1 and TX1 systems, shared memory accessed by both the CPU and GPU was allocated using the `cudaManagedMemory` function. However, the CUDA runtime prevents access to any of this memory from the CPU after a kernel starts executing on the GPU. We circumvented this issue by immediately unprotecting the appropriate regions of memory using `mprotect` each time after a GPU kernel is issued.

Platform	CPU	Cores	Freq. (GHz)	RAM	Accelerator	Cores	Freq. (MHz)	RAM	Linux Kernel	Acc Driver
K2	ARM Cortex-A15	4	1.4	2GB LPDDR3	C66 DSP	8	1200	Shared	4.4.32 *armhf*	OpenMP 4.0
TK1	ARM Cortex-A15	4	2.3	2GB LPDDR3	GK20A GPU	192	852	Shared	3.10.40 *armhf*	CUDA v6.5
TX1	ARM Cortex-A57	4	2.2	4GB LPDDR4	GM20B GPU	256	998	Shared	3.10.67 *aarch64*	CUDA v7.0
SANDY	Xeon E5-2665	2×8	2.4	128GB DDR3	K20m GPU (GK110)	2496	706	5GB	3.13.0	CUDA v7.0
HASWELL	Xeon E5-2620 v3	2×6	2.4	128GB DDR4	K80 GPU (GK210)	2496	875	12GB	3.16.0	CUDA v7.5

Table 6.1: Platforms use to experimentally validate energy usage model

The Energy Measurement Library (EML) provides a unified interface for both RAPL and NVML [Cabrera et al., 2015]. This library was used to measure the energy consumption of both HASWELL and SANDY. The energy measurement framework described in § 6.1 was used to measure energy on the LPSoC platforms.

6.5 Results and Analysis

The values of $R_c, R_g, P_{acig}, P_{icag}, P_{icg}$ were experimentally measured for each platform using DGEMM and SGEMM HPC kernels and are given in Table 6.2. During these measurements, care was taken to ensure that the CPU and accelerator frequencies were set to maximum using the *performance* frequency scaling governor for CPU and the explicit maximum frequency setting for the accelerator, only when they were in their active states. At all other times, CPU and accelerator frequencies were set to their minimum states. The *ondemand* CPU governor was used when idle on all systems. The GPU frequencies for idle states were set explicitly on TK1 and TX1. And on the Intel systems, *nvidia-smi* was used to set the GPU frequencies.

Table 6.2 presents the measured peak performances of both CPU and accelerator, i.e. R_c and R_g respectively. The sum of these two values, i.e. $R_c + R_g$ would represent the theoretical peak performance achievable from each platform using both CPU and accelerator. This table also presents the performance-optimal split values (SPLIT-GFLOPS) for each platform obtained using the adaptive work-partitioning approach. It is evident that for each platform, there exists a performance-optimal split that yields performance marginally less than the theoretical peak of $R_c + R_g$.

In fact, the value of f at which this performance-optimal split occurs corresponds to the point where $T_{ac} = T_{ag}$ i.e. both the CPU and accelerator finish work together and have no idle time during the computation. Equation 6.8 in § 6.2 defines this point and its value.

Also presented in Table 6.2 are the calculated values representing the condition to prove the existence of an energy-optimal work partition given by Equation 6.24 in § 6.2. In order for a work partition between the CPU and accelerator to be energy-optimal for a platform, the condition $\frac{P_{acig}-P_{icg}}{P_{icag}} < \frac{R_c}{R_g} < \frac{P_{acig}}{P_{icag}-P_{icg}}$ must hold.

Observing values of the lower and upper bounds of $\frac{R_c}{R_g}$ across all the platforms for both SGEMM and DGEMM, we find that only when performing DGEMM on the TX1 does the inequality hold i.e. $1.09 < 1.60 < 2.03$. This

Platform	Matrix Size	R_c (GFLOPS)	R_g (GFLOPS)	CPU SPLIT COLS	SPLIT GFLOPS	P_{acig} (W)	P_{icag} (W)	P_{icg} (W)	$\frac{P_{acig}-P_{icg}}{P_{icag}}$	$\frac{R_c}{R_g}$	$\frac{P_{acig}}{P_{icag}-P_{icg}}$
DGEMM											
K2	4096	9	24	784	29	17.00	14.39	6.0	0.76	0.38	2.03
TK1	4096	14	12	2320	25	12.47	6.06	1.55	1.80	1.17	2.76
TX1	4096	16	10	2528	24	11.76	8.38	2.59	1.09	1.60	2.03
SANDY	4096	303	726	1152	966	272.30	164.68	89.87	1.11	0.42	3.64
HASWELL	4096	345	736	1040	1191	189.18	138.63	71.65	0.85	0.47	2.82
SGEMM											
K2	4096	25	93	432	95	17.78	16.26	6.0	0.72	0.27	1.73
TK1	4096	36	201	512	230	12.99	11.47	1.55	0.99	0.18	1.31
TX1	4096	35	385	128	404	12.25	13.91	2.59	0.69	0.09	1.08
SANDY	4096	585	1668	1056	2109	257.72	148.78	89.87	1.13	0.35	4.37
HASWELL	4096	758	2314	960	2752	193.11	129.23	71.65	0.94	0.33	3.35

Table 6.2: Platform Characteristics: Performance and Power

is highlighted in green in table 6.2. For all other platforms, the inequality is not valid.

Given the measured system characteristics for each of these platforms, the energy usage model predicts that an energy-optimal split using both the CPU and accelerator would only exist for TX1 while performing DGEMM. For all other cases, using only the accelerator should always be energy-optimal.

In order to verify this prediction, the adaptive partitioning approach for GEMM outlined in § 5.4.1 was implemented and run on all platforms described in Table 6.1.

Figures 6.7-6.9 show the performance and energy consumption results of an experiment which used fixed size operand matrices of 4096×4096 for both SGEMM and DGEMM while dividing work based on columns of one input matrix given to the CPU. The domain of this experiment was $\texttt{CPU\_COLS} \in [0, 4096]$ where $\texttt{CPU\_COLS} = 0$ represents the case when all of the work is given to the accelerator and $\texttt{CPU\_COLS} = 4096$ represents when the CPU performs all the work.

The solid lines in figures 6.7-6.9 show performance achieved in GFLOPS for each work partition. The peaks or global maxima in these solid lines represent the performance-optimal work partitions.

The energy-to-solution values in Joules for each of these GEMM executions were measured using the apparatus described in § 6.1 and are represented by dotted lines in Figure 6.7-6.9. The energy-to-solution values predicted using the energy usage model for each system are also shown alongside using dashed lines.

Figure 6.7 presents results for K2. Figure 6.8 shows values for TK1 (in red) and TX1 (in green) while 6.9 shows values for SANDY (in red) and HASWELL (in green).

Consider first the K2 system. As predicted by the model, an energy-optimal split does not exist for both GEMM kernels. Predicted energy consumption values are almost identical to measured values. A performance-optimal split is found to exist for both GEMM kernels. It would however be energy-optimal to allocate all work to the DSP in each case.

For the TX1 system, as predicted using the energy model an energy-optimal split does indeed exist for the DGEMM work partition and its measured value is slightly larger than the energy model's predicted value. In fact, the measured values have a constant deviation from the modelled values. This deviation can be attributed to energy consumed by other system components such as DRAM.

While it is possible to factor in energy consumed by other components, the constant nature of their consumption provides no benefit in the model's eventual

goal of predicting the existence of an energy-optimal work partition. Hence, for simplicity, the model does not include these constant factors.

Each SGEMM execution for TK1 and TX1 and the DGEMM execution for TK1 yielded energy-to-solution values as expected by the energy usage model. For the Tegra systems, the TX1's Maxwell GPU vastly outperforms the TK1's Kepler GPU for SGEMM as expected due to the improved SP units. By comparison, the performance of TX1 DGEMM is dismal. The CPU performances across the TK1 and TX1 systems are roughly equivalent with the TX1 DGEMM performance being slightly improved.

Consider now the Intel systems. As predicted, using the CPU is not energy-optimal on either system. The measured energy-to-solution values however show an increasing deviation in the latter half of the graphs. Since CUDA unified memory managed by the runtime was used across benchmarks run on NVIDIA GPUs, the runtime itself was responsible for memory copies to and from GPU memory. Such explicit memory copies made by the CUDA runtime to and from the GPU across the PCIe bus, which are not hidden by overlapped computation are consistent with the observed deviation in the latter half of the graphs. Since the Tegra systems have shared physical memory, this was not an important factor for the TX1 and TK1. For systems with discrete GPU cards this does, however, have some impact (and could be factored into the model by proportionally reducing the effective computational rates R_c and R_g).

The relatively poor performance evident from the downward spikes on the HASWELL system, can be attributed to the unfavorable dimensions of the multiplication allocated to the GPU. The first half of the plots up to a split size of approximately 2048 is dominated by execution time on the GPU, while the second half is dominated by execution time on the CPU. Each spike in the first half corresponds to a load imbalance across the symmetric multi-processors (SM) on the K80 GPU (which has 15 SMs). The second half has no more spikes since the CPU execution time and energy provide the upper bound. This exaggerates existing challenges for cuBLAS's internal routines in their division of work into blocks, and load imbalance between the GPU's SMs.

6.5.1 Implications for Exascale Computing

It is of interest to consider the trade-offs between energy consumption and absolute performance required to achieve cost-effective exascale computing. Bergman et al. [2008] extrapolated the J/FLOP ratios of contemporary systems and suggested that a 20 fold increase in energy efficiency is required to meet the 20MW power budget for an exascale system. Dubé and Unit [2011] estimated the

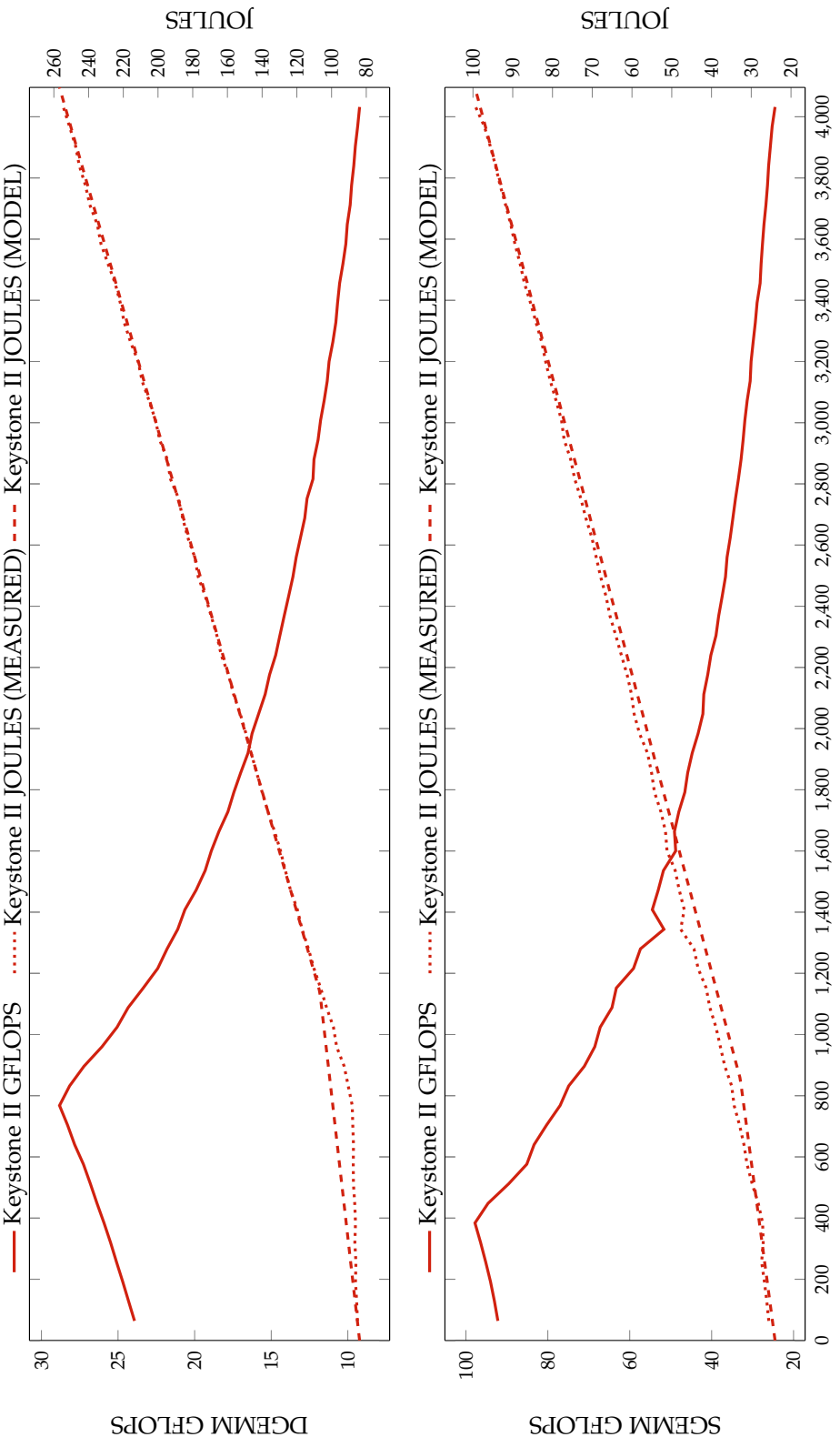


Figure 6.7: Partitioned GEMM: Performance and Energy - Keystone II

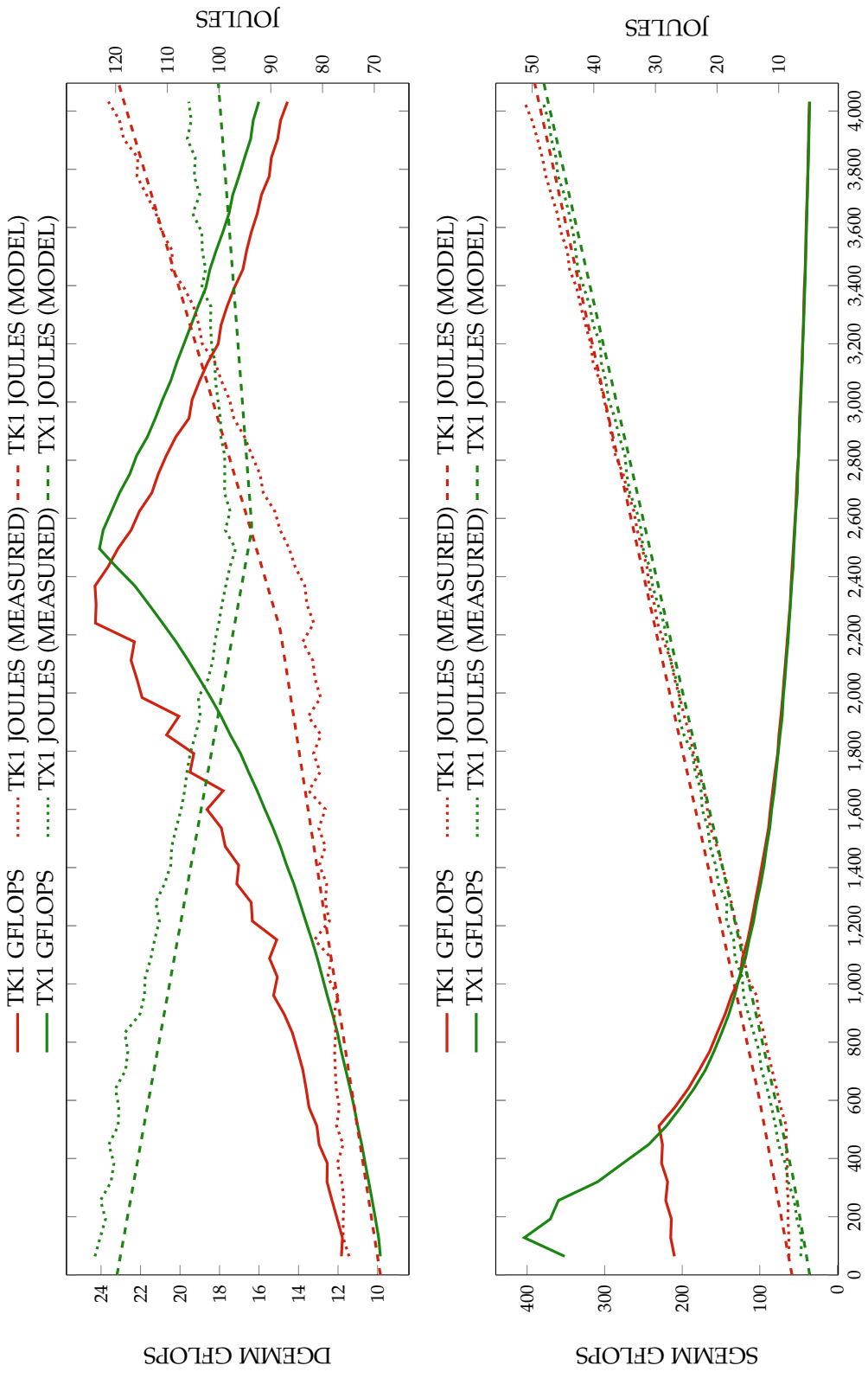


Figure 6.8: Partitioned GEMM: Performance and Energy - NVIDIA K1/X1

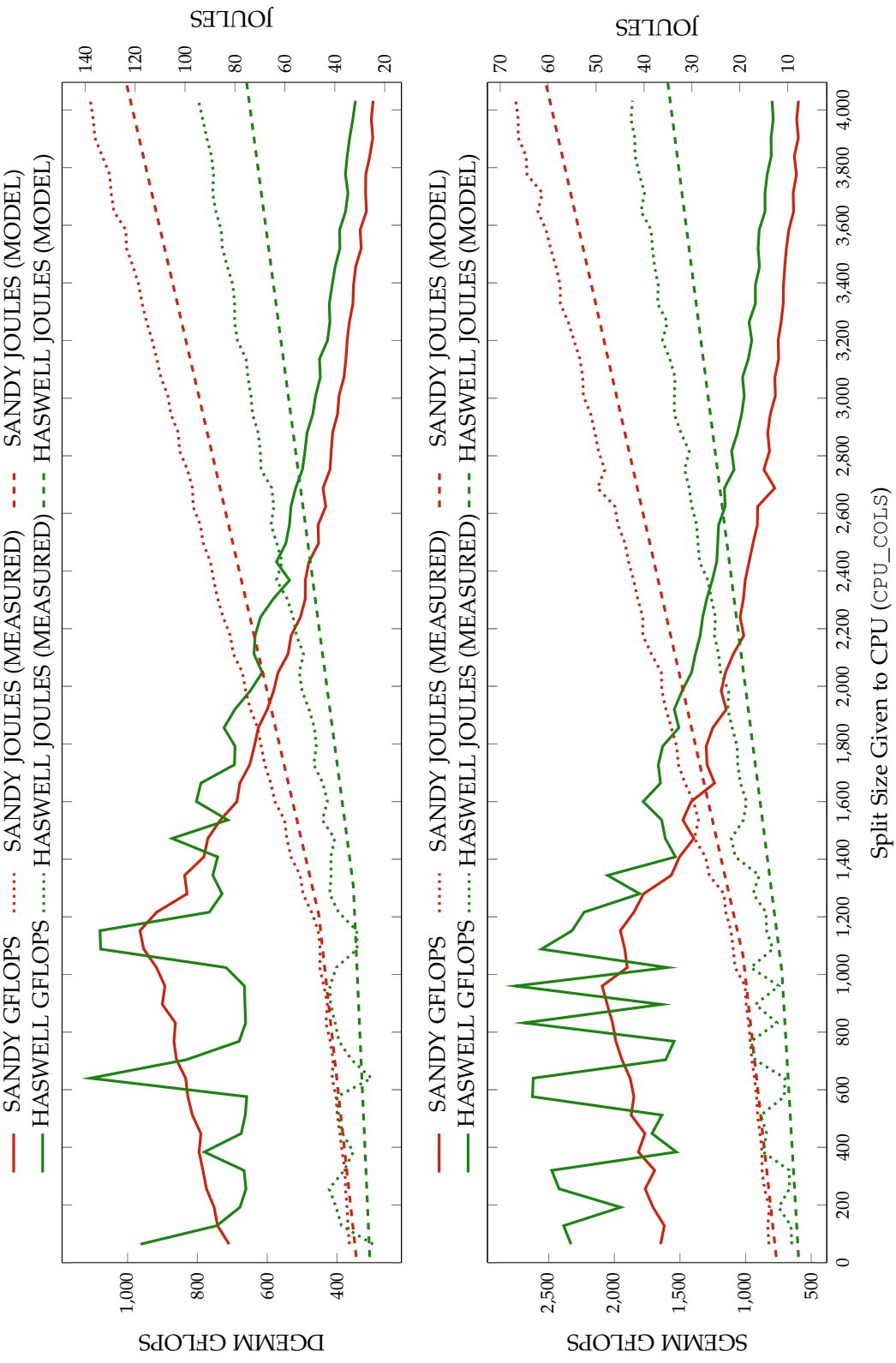


Figure 6.9: Partitioned GEMM: Performance and Energy - NVIDIA K20/K80

J/FLOP target for an exascale system with a power budget of 20 MW to be $5 - 10$ pJ/FLOP (DP) where 1 pJ = 10^{-12} J. Figures 6.10-6.12 shows the measured energy efficiency numbers across different matrix sizes on the K2, TX1 and HASWELL platforms using the J/FLOP metric.

Consider first the results from the K2 system presented in Figure 6.10. For matrix sizes above 2K, the energy efficiency stabilizes across all measurements. For SGEMM using only the DSP is most energy efficient with the best reading of 175 pJ/FLOP for the 8K case. In fact, for DGEMM is also most energy efficient on the DSP with the best reading of 610 pJ/FLOP for 8K matrices. Using ARM cores across both SGEMM and DGEMM reduces the energy efficiency. This fact was also predicted and validated by the energy model.

For results from the TX1 system given in Figure 6.11, the energy efficiency stabilizes across all measurements with input sizes above 1K. The GPU is the most energy efficient with the best reading of 28.4 pJ/FLOP for the 2K SGEMM experiment. For DGEMM, on the TX1 it is most energy efficient to use both the CPU and GPU together i.e. SPLIT. The best SPLIT DGEMM reading of 613 pJ/FLOP is observed for the 1K case. Again, the model correctly predicted an energy-optimal split for DGEMM.

Results from the HASWELL system given in Figure 6.12 indicate that larger matrix sizes computed only using the GPU are most energy efficient for SGEMM. An efficiency of 66.5 pJ/FLOP is measured for the 4K case. For DGEMM, It is evident that the SPLIT always gets the maximum performance. However, it is again more energy efficient to simply use the K80 GPU for DGEMM where the best reading of 189 pJ/FLOP is observed for the 8K case. Both results concur with the model's prediction.

Even though we expect the K80 GPU to be more energy efficient in all cases, in smaller cases the overhead of GPU kernel invocation, system calls, and threading on the CPU plays a large part in the energy measured, leading to the GPU approach consuming more energy than other approaches.

From these measurements, certain trade-offs are evident. Even though load balancing between CPU and accelerator on an LPSoC or other heterogeneous system may lead to maximum absolute performance, it does not imply optimal energy efficiency. The energy usage model also validates this claim.

However, there are cases where energy-optimality corresponds with performance-optimality when the CPU and accelerator are well balanced in performance. For e.g. the best SPLIT on the DGEMM 4K case on the TX1 gives a speedup of $3.34\times$ over just using the GPU while consuming $0.64\times$ the energy as compared to the GPU. This highlights the fact that the increase in performance obtained by using both CPU and accelerator on an LPSoC may not imply the equivalent decrease

in energy consumption.

Even though the K80 outperforms both the TX1 and K2 in terms of DP energy efficiency, the SP efficiency of the TX1 is marginally better than the K80, which is a state-of-the-art GPU accelerator for HPC.

In summary, while SP energy efficiency of contemporary LPSoC platforms are comparable to conventional HPC platforms today, for LPSoC platforms to be candidates for future exascale systems, their DP performance and energy efficiency must improve significantly. Compared to the exascale computing target of $5 - 10$ pJ/FLOP, the K2 achieved 610 pJ/FLOP, the TX1 achieved 613 pJ/FLOP and the HASWELL achieved 189 pJ/FLOP. Double-precision performance of future LPSoC systems requires an improvement of two orders of magnitude while maintaining similar energy consumption as measured in the K2 or TX1 systems to reach exascale energy efficiency. In comparison, a single order of magnitude performance improvement in attached accelerators such as the K80 GPU in the HASWELL system would enable future attached accelerators to achieve exascale efficiency.

6.6 Summary

This chapter presented an energy usage model for heterogeneous systems that allows prediction of an energy-optimal work partition between the CPU and accelerator. This model was validated using the widely used SGEMM and DGEMM matrix multiplication kernels on both LPSoC and conventional HPC systems with attached accelerators. A critique of the energy usage model was also provided. A highly accurate energy measurement system with a resolution similar to process time was demonstrated and used to collect fine-grained readings.

An adaptive work-partitioning scheme, for use on heterogeneous platforms was presented. Using adaptive partitioning, performance-optimal work partitions for GEMM kernels executing across five different heterogeneous systems were explored. The energy usage model predicted the existence of an energy-optimal work partition for one of the systems, the TX1 which also corresponded with its performance-optimal partition. The best SGEMM energy efficiency of 28.4 pJ/FLOP was observed on the TX1 while 189 pJ/FLOP was observed for DGEMM on the HASWELL system's K80 GPU. Trade-offs reflecting the performance-energy nexus of heterogeneous systems and their implications for future exascale systems were discussed.

A number of limitations were observed for work presented in this chapter.

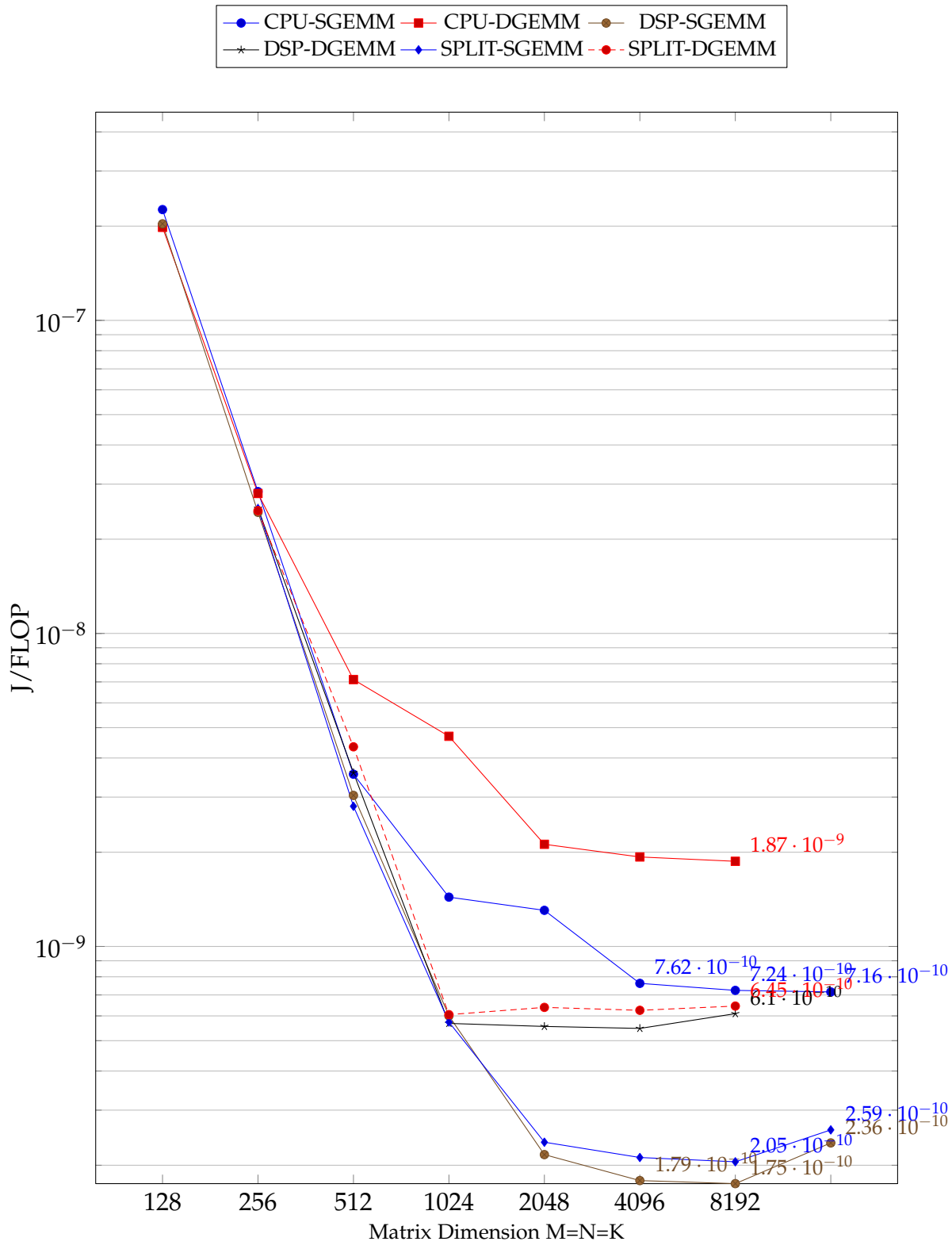


Figure 6.10: Energy Efficiency: TI Keystone II

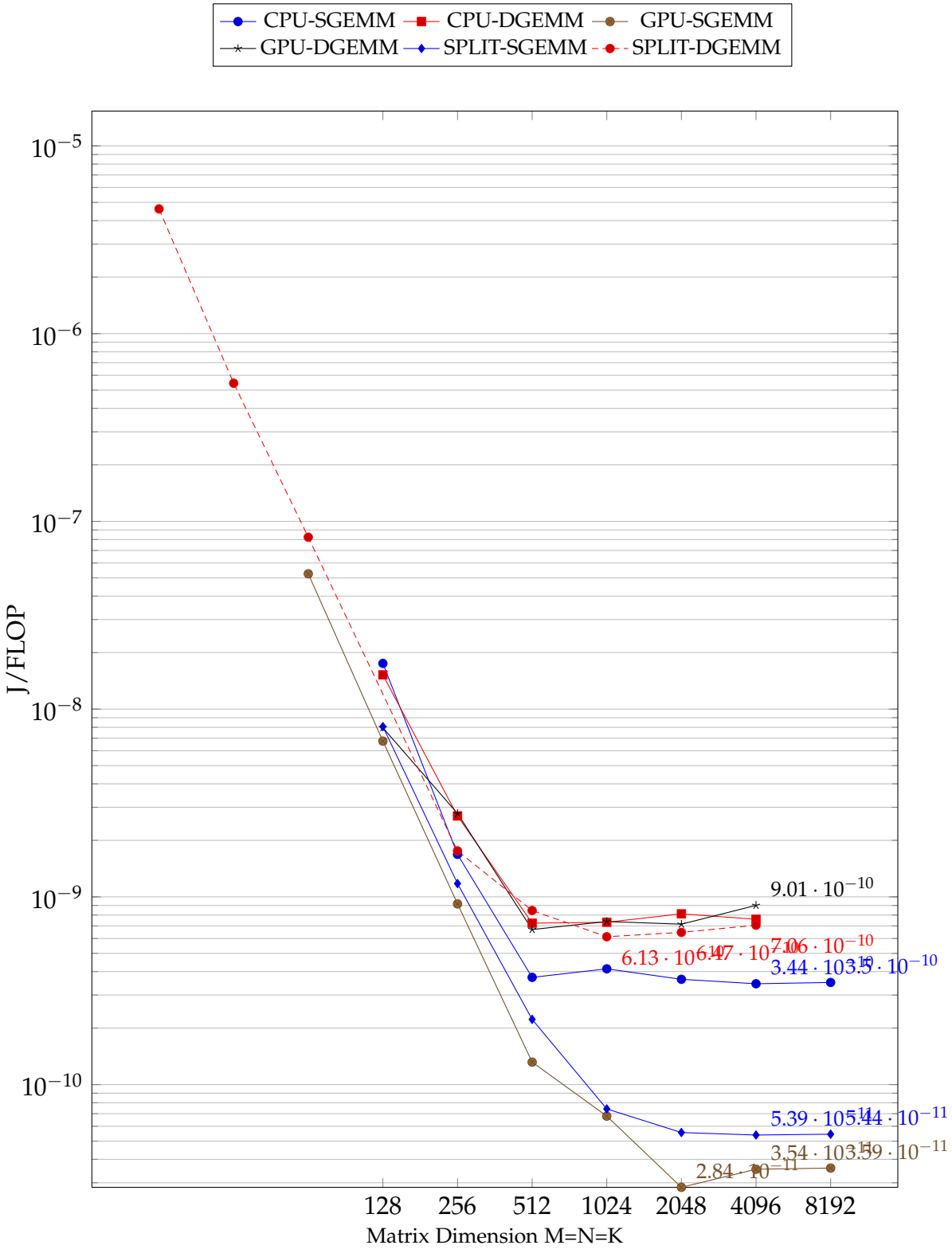


Figure 6.11: Energy Efficiency: TX1

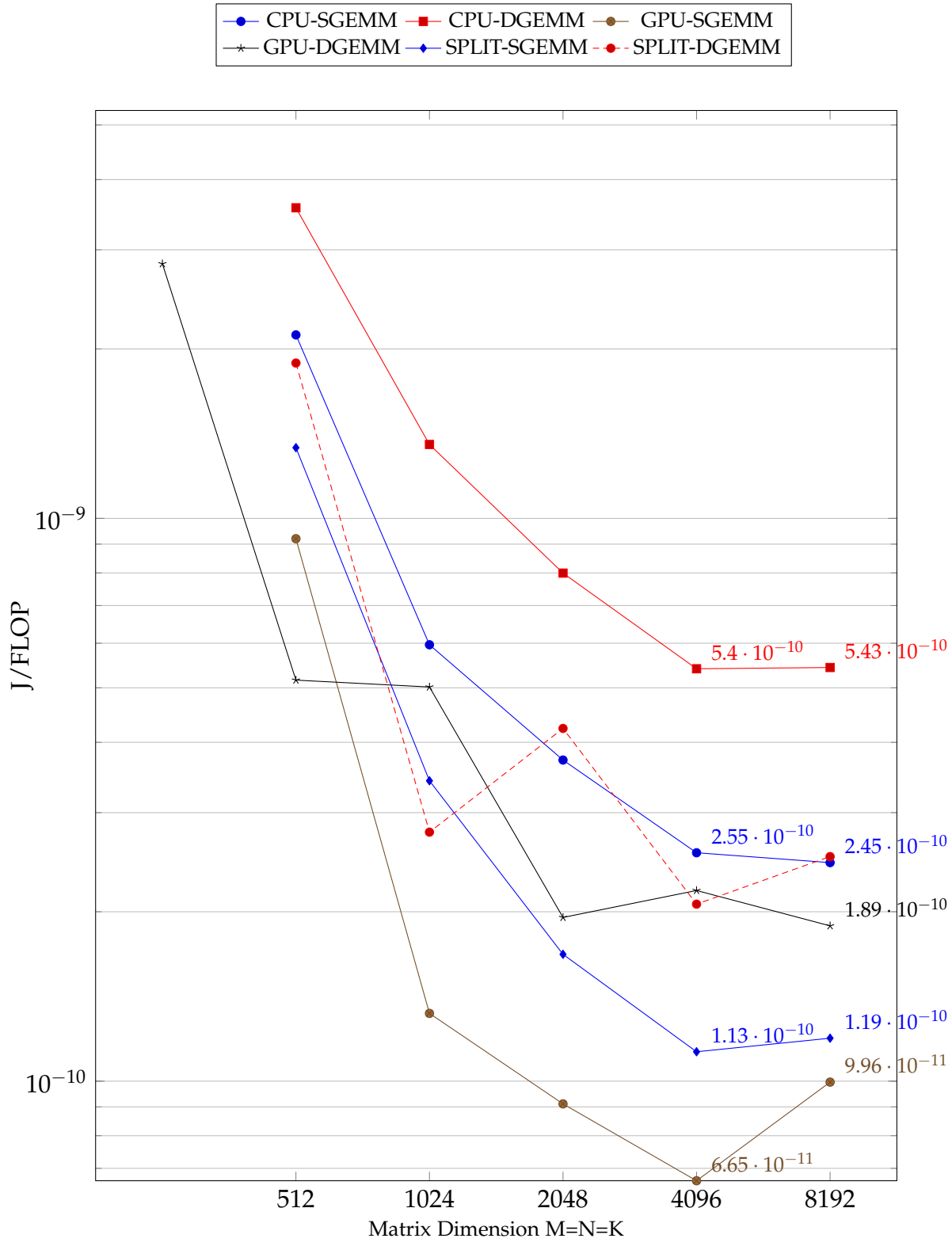


Figure 6.12: Energy Efficiency: Haswell + K80

These include the use of a single application benchmark, GEMM and experimentation across a small range of hardware platforms. Dynamic Voltage Frequency Scaling (DVFS) was not considered as only maximum and minimum frequency levels were used in active and idle states respectively for each platform. Further application of the energy usage model to larger systems with multiple attached accelerators was not demonstrated, even though it is theoretically possible. Input data was expected to be in place for the energy usage model to apply. Even though data transfers could implicitly be accounted for in the model as reduction in FLOP rates, as discussed in the model's critique, it would be beneficial to extend the model to explicitly account for data transfers. Each of these limitations could be addressed in future work.

Further validation of the energy usage model using other notable HPC applications and benchmark suites such as the NAS parallel benchmarks across more heterogeneous systems is of interest.

A novel application of the energy usage model would be to predict the target of dynamic work scheduling as performed in an OpenMP runtime. An energy efficient dynamic load balancing scheme using predicted values from the model could be useful for perhaps an OpenMP *energy* schedule clause similar to the guided schedule clause.

Moreover, how a running application might use information on energy usage to dynamically change its behavior and what this might mean in practice is an open question. For example, one objective may be to execute the application as fast as possible provided that the overall power usage of a given node does not exceed a certain total wattage. Tuning algorithms for optimal energy usage which will allow libraries like ATLAS to tune and produce best-performance and best-energy versions might be of interest.

Another scenario might factor in not just power usage but also energy cost, where the application code runs faster during times when energy is cheap e.g. from solar power generation. An extension of the energy measurement setup to support large systems with kW range power supply units would be of significant interest.

Chapter 7

Conclusions & Future Work

The overarching aim of the work presented in this thesis was to assess the viability of using an LPSoC processor designed for the embedded computing domain, namely the TI Keystone II (K2) LPSoC, for HPC applications. In so doing four major issues were considered.

1. Maximizing the K2 ARM CPU performance through improved code vectorization

2. Implementation and extension of the OpenMP programming model for the K2

3. Simultaneous use of ARM and DSP cores across multiple Keystone SoCs

4. An energy usage model to predict an energy-optimal work partition between CPU and accelerator on LPSoCs and heterogeneous systems in general

Chapter 3 looked at the ability of the compiler to exploit SIMD instructions on the ARM processor that resides on the K2. It was experimentally established that the use of compiler intrinsic functions on ARM CPUs yielded significantly higher performance compared to relying on compiler auto-vectorization. This was demonstrated using a set of OpenCV application benchmarks of varying complexity across nine ARM platforms including the K2. Comparisons were drawn with four Intel platforms where this was also found to be true. Use of ARM NEON intrinsics resulted in speed-up factors up to 13.88× compared to auto-vectorized code, while the corresponding speed-ups using Intel SSE2 intrinsics were up to 5.54×. The effectiveness of using ARM NEON intrinsics increased between generations of ARM CPUs with the highest increase of 3.7× being recorded between the 64-bit ARM Cortex-A57 and the previous generation 32-bit ARM Cortex-A15. While the A57 had the highest absolute performance

amongst all ARM CPUs evaluated, it had lower performance compared to the Intel IvyBridge processor.

Chapter 4 investigated the suitability of the OpenMP programming model for the K2. Evaluation of an alpha version of TI's bare-metal OpenMP runtime for C66x DSP cores indicated that the absence of hardware cache coherence on C66x DSP cores was a major performance limitation. A novel implementation of an OpenMP 4.0 runtime environment using OpenCL as a back-end was presented. Two performance optimizations were implemented to, i) to make use of contiguous shared memory and, ii) provide a new *local* OpenMP map-type to allocate buffers on fast DSP L2 scratchpad memory. This runtime was evaluated using an optimized implementation of the Level-3 BLAS Matrix Multiplication (GEMM) benchmark. Across eight DSP cores, SGEMM was measured at 110.11 GFLOPS, while DGEMM performed at 29.15 GFLOPS. Comparing performance of four DSP cores to four ARM cores, SGEMM was 6.7× faster and DGEMM was 2.6× faster on the DSP. These results indicated that use of the DSP accelerator on the K2 LPSoC is critical to achieve high performance in addition to using the ARM cores. The OpenMP 4.0 runtime environment was also found to be suitable to enable use of the DSP cores.

Chapter 5 looked at exploiting both ARM and DSP cores simultaneously on multiple K2 and Keystone I (K1) LPSoCs on the nCore BrownDwarf system. A proof-of-concept implementation and evaluation of a hybrid programming environment to enable simultaneous use of all processing elements on the Brown-Dwarf system was presented. Three components of this environment included: i) a message passing communication framework used to initiate program execution on K1 DSP cores from K2 ARM cores via Hyperlink, ii) creation of a single hybrid self-extracting fat binary used to initiate execution on a BrownDwarf node, and iii) use of a hybrid OpenMP/MPI programming model to simultaneously execute programs on all processing elements across multiple BrownDwarf nodes. This environment was evaluated using GEMM across four BrownDwarf nodes. Measured performance of 652.57 GFLOPS for SGEMM and 212.17 GFLOPS of DGEMM were observed. Bottlenecks in performance of the Hyperlink data transfer API were identified.

Chapter 6 investigated the energy-efficiency implications of using multiple devices simultaneously on heterogeneous systems such as K2 LPSoC. A high resolution energy measurement framework was created to allow measurement of energy consumption on LPSoC devices. An energy usage model for heterogeneous systems was created to predict a priori whether an energy-optimal work partition could exist between different devices. This model was experimentally validated using GEMM across three LPSoC systems, the K2, TK1 and TX1; and

two Intel based HPC systems with attached GPU accelerators. As predicted by the model, only on the TX1 system was there an energy-optimal work partition between CPU and GPU. On all other systems including the K2, running GEMM entirely on the accelerator was more energy-efficient. While the TX1 GPU had the best measured single-precision energy efficiency of 28.4 pJ/FLOP, the K80 GPU had the best double-precision efficiency of 189 pJ/FLOP. Compared to the estimated efficiency required for exascale of 5-10 pJ/FLOP (double-precision), contemporary LPSoC systems require at least two orders of magnitude improvement while contemporary attached accelerators are a single order of magnitude away from exascale. Results also demonstrated that the K80 and K40 GPUs had significantly higher raw performance compared to the TK1 and TX1 GPUs.

The work summarized above began in 2012 with the release of the K2. Since the launch of the K2, multiple other systems have emerged including the TK1 and TX1 with encompassing several improvements in manufacturing process technology. These newer systems were considered in this thesis alongside the K2 and represent a comparison across several generations of manufacturing technology. During the course of the above a number of lessons were learnt that merit comment. Also, there have been a number of hardware and software developments related to LPSoC systems and in HPC more generally.

7.1 Critique

Chapter 3 demonstrated the importance of using SIMD operations to achieve better performance. However, it was consistently observed that open-source compiler technology (gcc) across both ARM and Intel systems was unable to effectively exploit SIMD operations through auto-vectorization. With its increasing importance in contemporary many-core systems such as the Intel Knights Landing [Jeffers et al., 2016] with 512-bit SIMD instructions, and the Sunway TaihuLight's SW26010 [Fu et al., 2016] with 256-bit SIMD instructions, the ability of compilers to automatically exploit SIMD operations is becoming more important.

Even though using compiler intrinsic operations might lead to higher performance compared to auto-vectorization, there are two disadvantages to using them. Hand optimizing large application codes with intrinsics is a major undertaking and requires extra man-hours, increased cost and domain expertise. Once an application code is hand optimized, its portability to a different architecture

or system is reduced. Even transitioning to newer generations of the same architecture e.g. from ARMv7 to ARMv8 might require another round of hand optimization to maintain good performance. When code portability is a major concern, the quality of a vectorizing compiler emerges as a dominant factor for organizations trying to minimize costs when purchasing a new HPC system.

As evidenced by recent developments in the HPC space, the ARMv8 ISA is set to be adopted by major HPC manufacturers including HPE [Courtland, 2016], Fujitsu [Fujitsu, 2016] and Cray [Feldman, 2017] to be used in future generations of HPC systems. The primary source of performance of ARMv8 and future ARM ISA generations is its SIMD extensions, most notably the new Scalable Vector Extension (SVE) instructions [Stephens et al., 2017]. These recent developments bring work presented in Chapter 3 to the forefront and highlight its timeliness and importance. In fact, contributions from our work [Mitra et al., 2013] have been cited over 40 times since its publication and has sparked significant interest.

The rapid expansion of ARM based systems across several computing domains and its recent adoption into HPC systems by HPE and Fujitsu ensures that the ARM architecture will have continued support in HPC programming environments. Presumably, further advanced techniques to improve code vectorization using ARM NEON SIMD instructions will directly be implemented into open source compilers such as gcc thereby improving ease-of-use for HPC programmers. Availability of ARM optimized versions of critical scientific libraries such as BLAS, LAPACK and SCALAPACK is also expected to improve with the increasing adoption of ARM in HPC.

Chapter 4 and 5 demonstrated the importance of using the on-chip DSP cores on the K2 LPSoC to achieve good performance. However, the 32-bit registers on the C66x DSP cores had a major impact on double-precision performance. It was less than one fourth of the single-precision performance. This also had a negative impact on energy-efficiency, which was found to be 3.2$\times$ lower than the state-of-the-art K80 GPU. Scientific codes in the HPC community widely utilize double-precision calculations and the negative impact of using 32-bit processors is a barrier on adoption for many codes. Extension to 64-bit registers in future versions of this SoC would certainly be of interest to the HPC community.

Using GEMM we have observed that the DSP cores were approximately 12$\times$ faster for single-precision and 5$\times$ faster for double-precision compared to the ARM cores. Clearly, use of the DSP cores was critical for achieving high performance across the K2 LPSoC. Achieving such good performance for DSP GEMM kernels required use of C66x compiler intrinsic functions, in-depth knowledge of the SoC memory hierarchy, orchestration of data transfers using DMA co-

processors to overlap with computation and an efficient work partitioning strategy across multiple DSP cores. These optimizations took significant effort and this level of effort will be required for other HPC applications as well. In addition, these optimizations were entirely specific to the C66x DSP and are not portable across other accelerators such as GPUs.

The OpenMP 4.0 accelerator model allows a kernel to be offloaded to the DSP cores, however, simply offloading a naive GEMM kernel to the DSP would lead to poor performance. It was therefore necessary to optimize the GEMM kernel for the DSP cores. This significant optimization effort and loss of portability of the optimized DSP kernel somewhat defeats the objective of the OpenMP 4.0 specification. Unless more advanced techniques to automatically utilize complex memory hierarchies, vectorization, work partitioning across host and accelerator are invented, programming on-chip accelerators on LPSoC systems will remain difficult.

Optimizing GEMM for the DSP cores also revealed an important hardware feature critical to achieving good performance, the fast on-chip scratchpad memory (SRAM). The DSP cores allowed setting portions of L1 and L2 cache to SRAM. Having this explicit control of data movement between DDR RAM to MSMC SRAM to L2 SRAM and finally L1 SRAM was important to achieve good performance. Use of fast SRAM is not limited to embedded processors, it is in fact an integral feature of the Sunway TaihuLight SW26010 processor [Fu et al., 2016] which is the fastest supercomputer in the world as of June 2017. Similar to the Keystone DSP, the SW26010 processor has the ability to switch the fast on-chip memory between SRAM and cache modes. The slave cores on the SW26010 also require software-emulated cache coherence similar to the Keystone DSP cores. The on-chip memory was, however, used in user-controlled SRAM mode to achieve the 93 PFLOPS in the LINPACK benchmark.

As HPC programming models evolve into higher level abstractions such as OpenMP 4.0 and OpenACC [Wienke et al., 2012] compiler directives, ease of use and cross-platform portability emerge as key motivators of their evolution. This thesis demonstrates in Chapter 4 that LPSoC platforms such as the K2 keep pace with this evolution and are well suited for higher level HPC programming models such as OpenMP. The most recent OpenMP 4.5 [OpenMP ARB, 2015] specification released in November 2015 brought a number of additions specific to the use of accelerators. The notion of OpenMP tasks has now been applied to target regions with the *target nowait* directive. It essentially allows wrapping a target region within an OpenMP task with support for dependencies between different tasks. Unstructured data environments for accelerators are also supported with *target enter data* and *target exit data*. Both these additions of tasking

target regions and unstructured data readily apply to LPSoC systems such as the K2. The presence of shared contiguous memory between ARM and DSP cores enables this application.

Another significant addition to the OpenMP 4.5 specification was the introduction of device pointers. Using new device memory routines such as `omp_target_alloc`, it is now possible to allocate a buffer on an accelerator device and operate on that buffer from the host. One of the optimizations to the OpenMP 4.0 runtime environment presented in Chapter 4 demonstrated use of similar device memory routines such as `__malloc_ddr` to allocate and use buffers in device memory. Adoption of this optimization into the OpenMP 4.5 specification highlights the relevance and importance of work presented in this thesis.

Chapter 5 demonstrated that it is possible to simultaneously exploit both ARM and DSP cores on K1 and K2 LPSoCs on the nCore BrownDwarf system through the use of a hybrid programming environment. The gradient descent based adaptive work-partitioning algorithm developed in this work was used to partition GEMM computation across different processing elements on BrownDwarf nodes. The GEMM performance of K1 DSP cores compared to K2 DSP cores was significantly lower. This was primarily due the OpenMP runtime on K1 DSP cores occupying a large portion of L2 SRAM. This observation again brings to the forefront the importance of fast SRAM memory for heavily optimized application codes. Another limitation of the BrownDwarf system was the performance bottleneck observed across the Hyperlink interconnect between K2 and K1 LPSoCs. Further investigation into this Hyperlink performance bottleneck was not carried out, however, this would be of significant interest in future work.

Chapter 6 demonstrated the importance of understanding when optimal energy-efficiency might be achieved while using multiple processing elements on heterogeneous systems. Achieving optimal energy-efficiency in heterogeneous systems is of seminal importance to future exascale [Bergman et al., 2008; Shalf et al., 2011] supercomputers. While it was possible to significantly increase performance on LPSoC processors through simultaneous utilization of on-chip processing elements, this increase in performance was both theoretically and experimentally found to not always correspond with increased energy-efficiency.

Using the energy measurement framework and GEMM kernels, energy-efficiency of the K2 was experimentally measured and compared against current state-of-the-art HPC accelerators such as the NVIDIA K80 GPU. Energy consumed per FLOP on the Keystone II using the most energy-optimal work partition was found to be 2.6× higher for single-precision and 3.2× higher for double-

precision computation compared to a K80 GPU. Given the advances in processor manufacturing technology since the Keystone II was released in 2012, this is not a surprise. More recent LPSoC systems have already bridged this gap in energy-efficiency for single-precision computation. The TX1 system demonstrated higher single-precision energy-efficiency compared to the K80, and all other systems considered in this work. For double-precision computation, however, the K80 GPU consumed the least energy per FLOP.

Returning to the overarching aim, developing the programming environment and application software required for the K2 to succeed in the HPC space was non-trivial and required significant time. A number of issues relating to maximization of ARM CPU performance, the use of DSP accelerators and the energy efficiency implications of using both ARM and DSP processors were investigated and overcome.

During early research into the applicability of GPUs for HPC application codes, similar issues were encountered. Over time, and with increasing number of GPUs being available to researchers, mainly through their success in the gaming industry, these problems were overcome. Critical factors in the success of GPUs for HPC were their gaming driven demand and yearly turnaround with newer architecture iterations with higher performance especially by NVIDIA. Rapid improvements in computational performance were essential to be competitive in the gaming industry, a fact well suited for the HPC community. This similarity allowed GPUs to bridge the gap between two computing domains, gaming and HPC.

Even though LPSoC processors may be driven by mass-market adoption of smart devices, DSP accelerators have not enjoyed repeated architecture iterations to improve computational performance. The focus for manufacturers such as Texas Instruments regarding their floating-point DSP processors has consistently been towards minimizing power consumption for embedded computing applications. As a result, the 28nm Keystone II architecture, released in 2012 remains their most recent offering for HPC. Since 2012, multiple other LPSoC devices with vastly improved on-chip GPU accelerators such as the TK1 and TX1 have emerged. These systems widely reflect improvements in manufacturing process technology and increased energy-efficiency. As demonstrated, the TX1 GPU had the highest energy-efficiency for single-precision GEMM across both LPSoC and conventional Intel based HPC systems with attached GPUs.

With increasing heterogeneity of HPC systems, heavily optimized application codes may not be portable to future systems. Use of compiler directives and compiler assisted techniques could be the only practical method of maintaining portability of optimized code. With the ARM architecture being adopted

by the HPC community and the ability of manufacturers to license ARM intellectual property and tweak it for specific processor implementations, it may be possible for future ARM compilers to be co-designed with the ARM processor hardware. Co-design could result in a synergy between ARM compiler technology and ARM processors leading to highly efficient auto-vectorization and memory transfer orchestration.

Results obtained in this work indicate that conventional HPC systems today have an order of magnitude higher energy-efficiency for double-precision codes compared to contemporary LPSoC systems. While it may be beneficial to run single-precision codes on LPSoC systems, they are certainly not suited for double-precision codes compared to conventional systems. As a result, the Keystone II architecture from 2012 is unlikely to be part of future HPC systems. However, contributions of this thesis, especially relating to the simultaneous use of multiple processing elements and local scratchpad RAM, apply to heterogeneous LPSoC systems in general. It is an exciting time to be part of the HPC ecosystem with a new paradigm shift in HPC processor technology towards use of many low-power energy-efficient cores being well underway.

7.2 Future Work

Work presented in this thesis indicated the importance of using effectively using SIMD operations. Extending the study of SIMD operations on ARM systems to encompass recent additions to the ARM ISA such as the scalable vector extensions would be of interest. An experimental evaluation of SVE instructions with comparisons to Intel's AVX-512 SIMD instructions would also be valuable. Investigating the effectiveness of using the *simd* compiler directives introduced in OpenMP 4.0 to assist vectorization of loops and individual operations would be of great interest. In addition, expanding the range of benchmarks to encompass important scientific application codes such as GAMESS [Keipert et al., 2015] would be beneficial.

This thesis demonstrated the advantage of having shared physical memory available to on-chip compute devices on LPSoC processors. Work partitioning across different processing elements on an LPSoC was performed manually with the use of *target* offload directives in OpenMP. It would be interesting to extend the OpenMP programming model to consider this aspect and automatically distribute computation and orchestrate data movement across all compute devices sharing physical memory in an LPSoC rather than the programmer having to explicitly coordinate data movements and offload computation.

This would require understanding of the processing capabilities of each compute device, which could be performed in a runtime auto-tuning stage. While having OpenMP 4.0 directives to support separate address spaces is logical, future LPSoCs might have on-chip devices sharing the same MMU and therefore having the same address space, making room for OpenMP to automate work distribution amongst heterogeneous processing elements.

Implementation of the OpenMP 4.0 runtime environment on the Keystone II presented in this thesis showcased a functional mapping between OpenMP 4.0 constructs and the OpenCL library. Taking into consideration that the Keystone architecture is unlikely to be used in future HPC systems, it would however be of interest to apply the OpenMP 4.0 to OpenCL mapping on the BrownDwarf system to include the Keystone I. The implementation of the BrownDwarf communication framework demonstrated in this thesis provides a base layer and all necessary infrastructure for an OpenCL runtime library implementation targeted for Keystone I. Once an OpenCL library is implemented, it would be logical to extend the Keystone II OpenMP 4.0 runtime to offload computation to Keystone I SoCs. This would serve as an extended proof-of-concept for the OpenMP to OpenCL mapping encompassing both shared physical memory (between K2 ARM and K2 DSP) alongside distributed memory (between K2 ARM and K1 DSP). Extending the OpenMP 4.0 runtime environment to support the *teams* and *distribute* clauses to orchestrate computation amongst K2 ARM, K2 DSP and both K1 DSPs would also be worthwhile.

This thesis developed an environment that enabled detailed energy measurements for entire LPSoC systems at a resolution similar to that used in a profiler. It was also possible to start and stop energy measurement from within the application code being measured. With this capability one could envision an entirely new class of applications auto-tuned for energy-optimality. Any application which benefits from pre-tuning or runtime auto-tuning for performance could utilize this technique for an alternative goal to reduce energy consumption. This system could also be used to develop a hybrid load balancing low-level runtime system which would allocate work to the most energy-efficient compute devices during application execution. By extending the OpenMP schedule clause with a new target, *energy*, a programmer could specify that a kernel should automatically be distributed to the most energy-efficient compute device in a portable manner.

On the K2 and K1, work presented in this thesis clearly points to a major performance difference between single and double-precision on the DSP. Performance differences translate to energy per FLOP differences. There are two issues. First, having larger registers are likely to give more efficiency. The sec-

ond issue is providing the ability to divide these registers in multiple ways (via NEON/SSE SIMD instructions). Increasingly people are looking at using *just enough precision*. A good case is *deep learning* which is promoting the use of half precision. With their emerging popularity, deep learning application codes are starting to drive hardware manufacturing trends towards supporting half or even lower precision. HPC applications must be cognizant of this trend and explore possibilities around use of mixed or just enough precision.

Appendix A

SIMD benchmarks

A.1 Binary Image Thresholding

The algorithm used for binary thresholding is given below.

Algorithm 1 Pseudocode: Binary Image Threshold

```
for all pixels in Image do
    if pixel ≤ threshold then
        pixel ← threshold
    else
        pixel ← pixel
    end if
end for
```

A.2 Gaussian Blur

Due to the inherit property of the Gaussian function, this filter can be considered as the product of two 1D Gaussian filters, one for the horizontal direction, and one for the vertical direction. Each 1D Gaussian filter is then represented as

$$g(t) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2}$$

In this way, this operation can be implemented by convolving the image with two 1D filters successively, with the convolution operation defined as

$$I(x) = \sum_{t=-m}^{m} I(x-t) \times g(t) \tag{A.1}$$

where x is a pixel in image I, t denotes the filter entry index which is in the range of $[-m, m]$.

A.3 Sobel Filter

A Sobel filter is defined in the x and y directions by Shin [2010]

$$S_h = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad S_v = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{A.2}$$

The subscripts h and v denote horizontal and vertical filters respectively. If the image has N pixels, it may be partitioned into mini-blocks of 3×3 pixels so there would be $\frac{N}{9}$ partitions or thereabouts if N is not a factor of 9. For each pixel x, the filtered output is calculated by

$$I(x) = \sum_s S_s I_s$$

where s denotes the indexes of pixels and filter entries, and S can be one of the two filters define above. Pseudocode applicable to both the Gaussian and Sobel filters is given in Algorithm 2.

Algorithm 2 Pseudocode: Convolution Filtering

```
for all pixels I in Image do
    for all x pixels in width of filter S do
        for all y pixels in height of filter S do
            centre pixel I(*,*) += I(x,y) × S(x,y)
        end for
    end for
end for
```

Appendix B

Utility Functions for Keystone I DSP Application Code

```
#define EDMAMGR_MAX_CHAN_NUM    16
EdmaMgr_Handle channels[EDMAMGR_MAX_CHAN_NUM];

void edmamgr_channel_init(EdmaMgr_Handle* channels)
{
    int i;

    for (i = 0; i < EDMAMGR_MAX_CHAN_NUM; i++)
    {
        if (!(channels[i] = EdmaMgr_alloc(1)))
            printf("\tchan[%d] allocation failed\n", i);
    }
}

void edmamgr_channel_free(EdmaMgr_Handle* channels)
{
    int i;

    for (i = 0; i < EDMAMGR_MAX_CHAN_NUM; i++)
    {
        EdmaMgr_free(channels[i]);
    }
}
```

Figure B.1: K1 DSP: Initializing EDMA Channels

```
/* Memory barrier: Wait until data written to memory */
void mfence()
{
    // In order to accurately trace mfence instructions, a nop and mark
    // instruction must follow each mfence instruction
    asm (" mfence ");
    asm(" nop ");
    asm(" mark 0");
    asm (" mfence ");  // 2nd MFENCE as per Keystone I FAE Alert
    asm(" nop ");
    asm(" mark 0");

    // sprz332b.pdf, Advisory 24 - Require 16 NOPs after MFENCE after certain
    // cache coherency operations
    asm(" nop 9");
    asm(" nop 7");
}

// Write back and invalidate all L1 cache
void WbInvL1D (void)
{
    uint32_t lvInt = _disable_interrupts();
    CACHE_wbInvAllL1d(CACHE_NOWAIT);
    mfence();
    CSL_XMC_invalidatePrefetchBuffer();
    mfence();
    _restore_interrupts(lvInt);
    return;
}

void DSP_Create_L1D_SRAM()
{
    WbInvL1D();
    CACHE_wbInvAllL2(CACHE_WAIT);
    /* Set 16KB SRAM, 16KB 2-way cache
     * This makes the 1st half SRAM and 2nd half CACHE
     * SRAM: 00F0 0000 -> 00F0 4000 */
    CACHE_setL1DSize(CACHE_L1_16KCACHE);
}

void DSP_Restore_L1D_CACHE()
{
    WbInvL1D();
    /* Set 32KB 2-way cache */
    CACHE_setL1DSize(CACHE_L1_32KCACHE);
}

void DSP_wbInv_L1D() { WbInvL1D();}

void __cache_l2_flush()
{
    uint32_t lvInt = _disable_interrupts();
    CACHE_wbInvAllL2(CACHE_NOWAIT);
    mfence();
    CSL_XMC_invalidatePrefetchBuffer();
    mfence();
    _restore_interrupts(lvInt);

}
```

Figure B.2: K1 DSP: Cache operations used

```
#define MY_HW_TIM_IDX 0
extern cregister volatile uint32_t TSCL;
extern cregister volatile uint32_t TSCH;

int init_clock()
{
    volatile CSL_TmrRegsOvly *lvTimerRegPtr;
    lvTimerRegPtr = (CSL_TmrRegsOvly *)(CSL_TIMER_0_REGS +
                                        (CSL_TIMER_1_REGS -
                                           CSL_TIMER_0_REGS) *
                                           MY_HW_TIM_IDX);
    TSCL = 0; // start local timer
    return 0;
}

uint64_t __clock64()
{
    uint32_t low  = TSCL;
    uint32_t high = TSCH;
    return _itoll(high, low);
}
```

Figure B.3: K1 DSP: Cycle counter

Appendix C

Matrix Multiplication on a BrownDwarf node

The objective was to have GEMM kernels implemented for use on each processing element of a BrownDwarf node, i.e. Quad-core ARM, eight-core DSP on K2 and both the eight-core DSPs on K1 SoCs. These GEMM kernels on different processing elements must then be combined together to compute simultaneously on a single GEMM problem.

On the ARM processor, GEMM implementations provided by ATLAS [Whaley and Dongarra, 1998] (version 3.11.37) and BLIS [Note et al., 2013] were both built and evaluated. The BLIS library with the appropriate configuration for ARM Cortex-A15 was used as it performed better than auto-tuned ATLAS. The best BLIS configuration for parallel performance over four cores was achieved by splitting the j_r loop into four parts (BLIS_JR_NT $= 4$).

On the K2 DSP cores, the GEMM implementation using the OpenMP accelerator model runtime described and evaluated in chapter 4.2.5 was re-used. To the best of our knowledge, this implementation is the highest performant one available for K2 DSP cores.

Consider the K1 SoC on a BrownDwarf node. In order to implement SGEMM and DGEMM matrix for it, the first step was to understand the usable memory hierarchy on the K1. As initially implemented [Igual et al., 2012] for the K1, all four levels of the memory hierarchy are used. In order of increasing access latency, these are L1 and L2 scratchpad memory (SRAM) on each DSP core, MSMC scratchpad memory shared by all eight DSP cores and finally DDR heap memory shared by all cores.

Half of the L1 cache, i.e. 16KB, is re-configured to be L1 SRAM. Each C66x DSP on a K1 SoC has 512KB of L2. A portion of this memory, 128KB is configured as cache. The OpenMP runtime for the K1 provisions 171KB of usable scratchpad memory on L2 as it reserves the rest of the 384KB for core local heap and stack storage.

Using the same GotoBLAS approach implemented by [Igual et al., 2012], 16KB of L1 SRAM, 160KB of L2 SRAM on each DSP core and 1MB of MSMC SRAM on the SoC are reserved to store panels of packed input data elements for processing by the inner kernel.

Data transfer between the input matrices stored in DDR heap memory and these reserved memory segments is done using the asynchronous DMA engine (EDMA3) present on Keystone SoCs. This double buffering of data is performed in tandem with computation since the EDMA3 co-processor can operate independently alongside the DSP cores. Each DSP core uses two separate DMA channels to perform transfers between memory levels.

These independent GEMM kernels on each of the processing elements are then combined together on a single node using a mix of OpenMP on ARM, OpenMP accelerator model on K2 DSP and the communications framework for K1 SoCs.

Consider the use of the communications framework in the context of DGEMM. Figure C.1 illustrates the initialization of communication channels to each of the two K1 *Shannon* SoCs from code executing on the host ARM cores. A `dgemm_mem` structure is also defined. This contains locations of input matrices stored in K1 DRAM memory and details of the compute problem for an invocation of DGEMM on a K1.

Figure C.2 illustrates the steps involved in setting up a computation to run on a K1. First, the `malloc_msmc` and `malloc_ddr` function calls are used to create buffers in K1 MSMC SRAM and DRAM memory for storing job parameters and input/output matrices respectively.

Then input matrices are allocated and initialized on the ARM prior to the setup function being called. Once the K1 allocations are done, pointers to these buffers returned by the malloc calls are stored in the `job_parameters` structure. This is then copied to its buffer in K1 MSMC SRAM followed by the DMA transfers of the input matrices. The execution is now all set to begin on the K1.

Figure C.3 shows how a computation on K1 is invoked from the host ARM core. The `invoke_fn` function call can take two 32-bit payload items. Only one is required in this case. The address of the `job_parameters` structure in K1 MSMC SRAM is passed through. This allows the K1 DSP monitor to pick up the computation parameters from this structure and start the computation. This invoke operation is synchronous and waits for the K1 to finish execution and return a success or failure message to signal completion.

Figure C.4 demonstrates how results in buffer C are copied back to ARM DDR memory from K1 memory after K1 computation is finished. In addition, the time taken to perform the computation is calculated on the K1 using a hard-

```
typedef struct
{
    uint32_t    param;
    uint32_t    time;
    uint32_t    M;
    uint32_t    N;
    uint32_t    K;
    uint32_t    A;          // double*
    uint32_t    B;          // double*
    uint32_t    C;          // double*
    uint32_t    omp_threads;
} dgemm_mem;

Commchannel* chan_shn0;
Commchannel* chan_shn1;

dgemm_mem* shn0_params;
dgemm_mem* shn1_params;

void init_k1_socs(int debug)
{
    chan_shn0 = new Commchannel(SOC_SHN0, debug);
    chan_shn1 = new Commchannel(SOC_SHN1, debug);

    shn0_params = new dgemm_mem;
    shn1_params = new dgemm_mem;
}
```

Figure C.1: K2 ARM: Initializing communications channels to K1

```
void cblas_dgemm_k1_setup(const char* soc,
                          double* A,
                          double* B,
                          int M,
                          int N,
                          int K,
                          int nthreads)
{
    Commchannel* shn;
    dgemm_mem* job_parameters;

    if      (strcmp(soc, SOC_SHN0) == 0)
    {shn = chan_shn0; job_parameters = shn0_params;}
    else if (strcmp(soc, SOC_SHN1) == 0)
    {shn = chan_shn1; job_parameters = shn1_params;}

    /* Allocate job parameter storage area on K1 MSMC */
    uint32_t shn_param_storage = shn->malloc_msmc(sizeof(dgemm_mem));
    uint32_t shn_time_storage = shn->malloc_msmc(sizeof(double));

    /* Allocate storage area on K1 DDR for input and output data */
    uint64_t shn_A_storage  = shn->malloc_ddr(M*K*sizeof(double));
    uint64_t shn_B_storage  = shn->malloc_ddr(K*N*sizeof(double));
    uint64_t shn_C_storage  = shn->malloc_ddr(M*N*sizeof(double));

    if (!shn_A_storage || !shn_B_storage || !shn_C_storage)
    {
        cout << "K1 memory allocation failed on " << soc << endl;
        exit(-1);
    }

    /* Populate job parameters */
    job_parameters->param        = shn_param_storage;
    job_parameters->time         = shn_time_storage;
    job_parameters->M            = (uint32_t) M;
    job_parameters->N            = (uint32_t) N;
    job_parameters->K            = (uint32_t) K;
    job_parameters->A            = shn_A_storage;
    job_parameters->B            = shn_B_storage;
    job_parameters->C            = shn_C_storage;
    job_parameters->omp_threads  = nthreads;

    /* Copy job parameters to K1 */
    shn->copy_to((char*)job_parameters,
                 (uint64_t) shn_param_storage,
                 sizeof(dgemm_mem)
                 );

    /* Copy input data to K1 */
    shn->dma_to(A, shn_A_storage, M*K*sizeof(double));
    shn->dma_to(B, shn_B_storage, K*N*sizeof(double));
}
```

Figure C.2: K2 ARM: Setting up computation parameters for K1 DGEMM

```
void cblas_dgemm_k1_compute(const char* soc)
{
    Commchannel* shn;
    dgemm_mem* job_parameters;

    if      (strcmp(soc, SOC_SHN0) == 0)
    {shn = chan_shn0; job_parameters = shn0_params;}
    else if (strcmp(soc, SOC_SHN1) == 0)
    {shn = chan_shn1; job_parameters = shn1_params;}

    /* Invoke dgemm job on K1 and provide address of job parameters */
    shn->invoke_fn(job_parameters->param, 0);
}
```

Figure C.3: K2 ARM: Invoking K1 DGEMM

ware cycle counter present on the K1. This compute time is also retrieved to calculate the overhead of the communication framework.

```
void cblas_dgemm_k1_results(const char* soc, double* C, int M, int N,
    double* device_time)
{
    Commchannel* shn;
    dgemm_mem* job_parameters;

    if      (strcmp(soc, SOC_SHN0) == 0)
    {shn = chan_shn0; job_parameters = shn0_params;}
    else if (strcmp(soc, SOC_SHN1) == 0)
    {shn = chan_shn1; job_parameters = shn1_params;}

    /* Copy results from K1 to local dst */
    shn->dma_from(job_parameters->C, C, M*N*sizeof(double));
    /* Copy device compute time */
    shn->copy_from(job_parameters->time, device_time, sizeof(double));
}
```

Figure C.4: K2 ARM: Retrieving K1 DGEMM Results

Figure C.5 shows how the buffers allocated on the K1 for the DGEMM computation are released. The `free_msmc` and `free_ddr` function calls are used to release memory reserved for these buffers to the K1 heap for further use. Finally, figure C.6 shows the closing of the communication channels to the K1 SoCs. This concludes the K2 ARM usage of the communication framework.

```
void cblas_dgemm_k1_cleanup(const char* soc)
{
    Commchannel* shn;
    dgemm_mem* job_parameters;

    if      (strcmp(soc, SOC_SHN0) == 0)
    {shn = chan_shn0; job_parameters = shn0_params;}
    else if (strcmp(soc, SOC_SHN1) == 0)
    {shn = chan_shn1; job_parameters = shn1_params;}

    /* Free K1 resources */
    shn->free_msmc(job_parameters->param);
    shn->free_msmc(job_parameters->time);
    shn->free_ddr(job_parameters->A);
    shn->free_ddr(job_parameters->B);
    shn->free_ddr(job_parameters->C);
}
```

Figure C.5: K2 ARM: Releasing K1 resources

```
void close_k1_socs()
{
    chan_shn0->close();
    chan_shn1->close();

    delete chan_shn0;
    delete chan_shn1;
    delete shn0_params;
    delete shn1_params;
}
```

Figure C.6: K2 ARM: Closing communications channel to K1

Consider the K1 DSP implementation of DGEMM. Figure C.7 illustrates the function `__k1_fn_call()` which is run when the computation is invoked from the ARM host program.

This function begins with initializing the 16 EDMA channels to be used on the DSP cores using the `edmamgr_channel_init()` function. This initialization process allocates memory for the EDMA channels as shown in § B.1.

Following this, the hardware cycle counter is initialized using the `init_clock()` function. This initializes the TSCL and TSCH timer registers for local cycle counting as shown in § B.3.

Once the DMA channels and cycle counter is initialized, the job parameters are extracted from the payload address sent from the ARM. This address points to the populated `dgemm_mem` structure which has already been copied over from the ARM. In between memory read/write operations, the L1 and L2 caches are flushed to maintain memory consistency.

After the job parameters are extracted, a 1MB buffer is allocated on MSMC for use during DGEMM computation. The `cblas_dgemm_omp()` function is then called which initiates the DGEMM computation across all eight DSP cores using the OpenMP runtime.

Figure C.8 illustrates the dgemm kernel launch using OpenMP. Prior to the launch, the L2 SRAM are of 171KB is initialized. This is done by placing a 64-byte aligned buffer `l2_buf` in a linker section `.mem_l2` using a compiler pragma in line 4. During compilation, the K1 linker places this section in L2 memory.

Once the function starts, the cycle counter is started using the `__clock64()` function. An OpenMP parallel region is then created. Provided that `omp_threads=8`, each DSP core is assigned a single thread. Each core then proceeds to initialize its own L2 heap structure using the `__heap_init_l2()` function and then reserve a 160KB buffer in that heap for storing DGEMM input panels.

The L1 cache is then re-configured to be half cache, half L1 SRAM. The start address of the L1 SRAM is already known to be 0x00F00000 for this configuration. At this point the core-local DGEMM kernel is invoked and it is passed the L1, L2, MSMC buffer addresses that it can use along with its thread id. The DMA channels allocated for it are also passed in.

Once the core-local kernel completes, the L1 cache is reset to 32KB and the OpenMP parallel region ends. The cycle counter is again used to calculate the elapsed cycles and given that the DSP is running at 1.2 Ghz, the time elapsed is calculated and returned.

Various cache operations and L1 reconfiguration operations are documented in § B.2.

```
#define BYTES_1024KB                (0x100000) /* 1024 KB total = 128 KB per core */

void __k1_fn_call(void *payload0, void *payload1)
{
    printf("APP:\tStarting dgemm job\n");
    printf("APP:\tReceived ->\tPayload0: 0x%p\n\t\t\tPayload1: 0x%p\n", payload0,
        payload1);
    edmamgr_channel_init(channels);
    init_clock();
    /* payload0 contains address of job parameter structure */
    dgemm_mem *job_parameters = (dgemm_mem *) ((uint32_t)payload0);
    CACHE_invL2((void *)job_parameters, sizeof(dgemm_mem), CACHE_WAIT);
    DSP_wbInv_L1D();

    /* Extract job parameters */
    double *time       = (double*)job_parameters->time;
    int     M          = (int   ) job_parameters->M   ;
    int     N          = (int   ) job_parameters->N   ;
    int     K          = (int   ) job_parameters->K   ;
    double  *A         = (double*) job_parameters->A  ;
    double  *B         = (double*) job_parameters->B  ;
    double  *C         = (double*) job_parameters->C  ;
    int     omp_threads= (int   ) job_parameters->omp_threads;

    DSP_wbInv_L1D();
    __cache_l2_flush();

    /* Allocate MSMC SRAM buffer */
    int msmc_size = BYTES_1024KB;
    double* MSMC_buf = __memalign_msmc(4096, msmc_size);

    /* Run job */
    *time = cblas_dgemm_omp(M,
                    N,
                    K,
                    A,
                    B,
                    C,
                    MSMC_buf,
                    msmc_size,
                    omp_threads
                    );

    DSP_wbInv_L1D();
    __cache_l2_flush();

    printf("APP:\tdgemm done\n");
    __free_msmc(MSMC_buf);
    edmamgr_channel_free(channels);

    return 0;
}
```

Figure C.7: K1 DSP: Extracting job parameters to setup DGEMM

```
#define L1D_SRAM_START              (0x00F00000)
#define DSP_L2SRAM_HEAP_SIZE        (0x2AC00) /* 171 KB per core */
#define DSP_L2SRAM_ALLOC_SIZE       (0x28000) /* 160 KB per core */
#pragma DATA_SECTION(l2_buf, ".mem_l2")
#pragma DATA_ALIGN(l2_buf, 64)
char l2_buf[DSP_L2SRAM_HEAP_SIZE];

double cblas_dgemm_omp(int M, int N, int K,
                       double *A,
                       double *B,
                       double *C,
                       double *MSMC_buf,
                       int msmc_size,
                       int omp_threads)
{
        uint64_t start = 0, end = 0;
        start = __clock64();
        #pragma omp parallel default(shared) num_threads(omp_threads)
        {
            int _lda = M;
            int _ldb = K;
            int _ldc = CORE_PROCESS_ROWS*((M+(CORE_PROCESS_ROWS-1))/CORE_PROCESS_ROWS);
            int tid, mLocal;
            double* pL1 = (double*) L1D_SRAM_START;
            __heap_init_l2(l2_buf, DSP_L2SRAM_HEAP_SIZE);
            double *pL2 = __malloc_l2(DSP_L2SRAM_ALLOC_SIZE);
            int nthreads = omp_get_num_threads();
            int mRemaining = 0;
            tid = omp_get_thread_num();
            mLocal = (nthreads > M ? 1 : M/nthreads);
            if (tid == nthreads-1)
            {
                if (M % nthreads != 0)
                    mRemaining = M % mLocal;
            }
            /* Configure L1D on each core to have 16KB SRAM and 16 KB Cache */
            DSP_Create_L1D_SRAM();
            dgemm(mLocal + mRemaining, N, K,
                    A + mLocal*tid, _lda,
                    B, _ldb,
                    C + mLocal*tid, _ldc,
                    pL1, pL2, MSMC_buf, tid,
                    channels[tid], channels[tid+8]
                 );
            /* Restore L1D Cache config on each core to entire 32KB Cache */
            DSP_Restore_L1D_CACHE();
        }
        end = __clock64();
        uint64_t elapsed = end - start;
        double time = ((double)elapsed) * (1.0/1.2) * 1.0e-9;
        printf("APP:\t cycles: %lld, time: %fs\n", elapsed, time);
        return time;
}
```

Figure C.8: K1 DSP: Calling the OpenMP DGEMM kernel

Bibliography

Android NDK. http://developer.android.com/tools/sdk/ndk/index.html. Accessed: 14/10/2012. (cited on page 56)

OpenCV For Android. http://code.opencv.org/projects/opencv/wiki/OpenCV4Android. Accessed: 14/10/2012. (cited on page 56)

SPRS866: 66AK2H12/06 Multicore DSP+ARM Keystone II System-on-Chip (SoC). Texas Instruments Literature. (cited on page 17)

2004. *SPRU423D: DSP/BIOS user's guide*. Texas Instruments Literature. (cited on pages 33 and 42)

2010. *SPRUGO6A: SYS/BIOS inter-processor communication (IPC) and I/O user's guide*. Texas Instruments Literature. (cited on page 42)

2013. Hardkernel ODROID. Technical report. Accessed: 22/01/2013. (cited on page 56)

Agullo, E.; Demmel, J.; Dongarra, J.; Hadri, B.; Kurzak, J.; Langou, J.; Ltaief, H.; Luszczek, P.; and Tomov, S., 2009. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. In *Journal of Physics: Conference Series*, vol. 180, 012037. IOP Publishing. (cited on page 43)

Ahmad, A.; Ali, M.; South, F.; Monroy, G. L.; Adie, S. G.; Shemonski, N.; Carney, P. S.; and Boppart, S. A., 2013. Interferometric synthetic aperture microscopy implementation on a floating point multi-core digital signal processer. In *SPIE BiOS*, 857134–857134. International Society for Optics and Photonics. (cited on page 41)

Ali, M.; Stotzer, E.; Igual, F. D.; and van de Geijn, R. A., 2012. Level-3 BLAS on the TI C6678 multi-core DSP. In *Computer Architecture and High Performance Computing (SBAC-PAD), IEEE 24th International Symposium on*, 179–186. IEEE. (cited on pages 14, 41, 99, and 103)

ARM Limited, 2009. *Introducing NEON^{TM} - Development Article*. (cited on pages 17, 26, and 28)

ARM Limited, 2011. *ARM Architecture Reference Manual - ARM v7-A and ARM v7-R edition Errata markup*. (cited on pages 27 and 70)

ARM Limited, 2012. *ARM v8-A Instruction Set Overview*. (cited on page 26)

ARM Ltd., 2013. ARM Annual Report 2013. http://ir.arm.com/phoenix.zhtml?c=197212&p=irol-reportsannual. Accessed: 20/10/2014. (cited on page 2)

Attig, N.; Gibbon, P.; and Lippert, T., 2011. Trends in supercomputing: The european path to exascale. *Computer Physics Communications*, 182, 9 (2011), 2041–2046. (cited on page 1)

Augonnet, C.; Thibault, S.; and Namyst, R., 2010. Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par 2009–Parallel Processing Workshops*, 56–65. Springer. (cited on page 44)

Augonnet, C.; Thibault, S.; Namyst, R.; and Wacrenier, P.-A., 2011. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23, 2 (2011), 187–198. (cited on page 43)

Ayguadé, E.; Badia, R. M.; Bellens, P.; Cabrera, D.; Duran, A.; Ferrer, R.; Gonzàlez, M.; Igual, F.; Jiménez-González, D.; Labarta, J.; et al., 2010. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38, 5-6 (2010), 440–459. (cited on page 42)

Ayguade, E.; Badia, R. M.; Cabrera, D.; Duran, A.; Gonzalez, M.; Igual, F.; Jimenez, D.; Labarta, J.; Martorell, X.; Mayo, R.; et al., 2009. A proposal to extend the openmp tasking model for heterogeneous architectures. In *Evolving OpenMP in an Age of Extreme Parallelism*, 154–167. Springer. (cited on page 42)

Balaprakash, P.; Tiwari, A.; and Wild, S. M., 2013. Multi objective optimization of hpc kernels for performance, power, and energy. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, 239–260. Springer. (cited on page 46)

Barker, J. and Bowden, J., 2013. Manycore Parallelism through OpenMP. In *OpenMP in the Era of Low Power Devices and Accelerators*, 45–57. Springer. (cited on page 42)

Beaumont, O.; Boudet, V.; Rastello, F.; and Robert, Y., 2001. Matrix Multiplication on Heterogeneous Platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 12, 10 (2001), 1033–1051. (cited on pages 43 and 44)

Becker, D. J.; Sterling, T.; Savarese, D.; Dorband, J. E.; Ranawak, U. A.; and Packer, C. V., 1995. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, vol. 95. (cited on page 2)

Becker, J.; Huebner, M.; and Ullmann, M., 2003. Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-Offs and Limitations. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, 283–288. IEEE. (cited on page 45)

Bedard, D.; Lim, M. Y.; Fowler, R.; and Porterfield, A., 2010. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the*, 479–484. IEEE. (cited on page 45)

Bentmar Holgersson, S., 2012. Optimising IIR Filters Using ARM NEON. (2012). (cited on page 40)

Bergman, K.; Borkar, S.; Campbell, D.; Carlson, W.; Dally, W.; Denneau, M.; Franzon, P.; Harrod, W.; Hill, K.; Hiller, J.; et al., 2008. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15 (2008). (cited on pages 1, 155, and 170)

Beyer, J. C.; Stotzer, E. J.; Hart, A.; and de Supinski, B. R., 2011. Openmp for accelerators. In *OpenMP in the Petascale Era*, 108–121. Springer. (cited on page 42)

Blake, A., 2012. *Computing the fast Fourier transform on SIMD microprocessors*. Ph.D. thesis, University of Waikato. (cited on pages 40 and 41)

Bradski, G. and Kaehler, A., 2008. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, Cambridge, MA. (cited on page 50)

Brunschen, C. and Brorsson, M., 2000. OdinMP/CCp - a portable implementation of OpenMP for C. *Concurrency - Practice and Experience*, 12, 12 (2000), 1193–1203. (cited on page 32)

Bull, J. M.; Reid, F.; and McDonnell, N., 2012. A microbenchmark suite for openmp tasks. In *OpenMP in a Heterogeneous World*, 271–274. Springer. (cited on page 77)

Cabrera, A.; Almeida, F.; Arteaga, J.; and Blanco, V., 2015. Measuring Energy Consumption Using EML (Energy Measurement Library). *Computer Science-Research and Development*, 30, 2 (2015), 135–143. (cited on page 152)

Cabrera, D.; Martorell, X.; Gaydadjiev, G.; Ayguade, E.; and Jiménez-González, D., 2009. Openmp extensions for fpga accelerators. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*, 17–24. IEEE. (cited on page 42)

Catalán, S.; Herrero, J. R.; Igual, F. D.; Rodríguez-Sánchez, R.; and Quintana-Ortí, E. S., 2015. Multi-threaded dense linear algebra libraries for low-power asymmetric multicore processors. *arXiv preprint arXiv:1511.02171*, (2015). (cited on page 44)

Chapman, B.; Huang, L.; Biscondi, E.; Stotzer, E.; Shrivastava, A.; and Gatherer, A., 2009. Implementing openmp on a high performance embedded multicore mpsoc. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 1–8. IEEE. (cited on page 41)

Chen, T.; Raghavan, R.; Dale, J.; and Iwata, E., 2007. Cell broadband engine architecture and its first implementation and performance view. *IBM Journal of Research and Development*, 51, 5 (2007), 559–572. (cited on page 26)

Courtland, R., 2016. Can hpe's "the machine" deliver? *IEEE Spectrum*, 53, 1 (2016), 34–35. (cited on pages 3 and 168)

Cramer, T.; Schmidl, D.; Klemm, M.; and an Mey, D., 2012. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. (2012), 38–44. (cited on page 42)

Dagum, L. and Menon, R., 1998. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5, 1 (1998), 46–55. (cited on page 30)

Diamos, G. F. and Yalamanchili, S., 2008. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, 197–200. ACM. (cited on page 44)

Donfack, S.; Tomov, S.; and Dongarra, J., 2014. Dynamically Balanced Synchronization-Avoiding LU Factorization with Multicore and GPUs. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, 958–965. IEEE. (cited on page 43)

Dongarra, J. and Luszczek, P., 2012. Anatomy of a Globally Recursive Embedded LINPACK Benchmark. *IEEE High Performance Extreme Computing Conference (HPEC)*, (2012). (cited on pages 1 and 2)

Dubé, N. and Unit, H. B., 2011. True Sustainability, the Path to a Net-Zero Datacenter: Energy, Carbon, Water. In *Proceedings of the 2011 workshop on Energy Efficiency: HPC System and Datacenters*, 201–236. ACM. (cited on page 155)

Eichenberger, A. E.; O'Brien, J. K.; O'Brien, K. M.; Wu, P.; Chen, T.; Oden, P. H.; Prener, D. A.; Shepherd, J. C.; So, B.; Sura, Z.; et al., 2006. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45, 1 (2006), 59–84. (cited on page 42)

Fatica, M. and Phillips, E., 2014. Synthetic Aperture Radar imaging on a CUDA-enabled mobile platform. (2014). (cited on page 40)

Feldman, M., 2017. Cray to Deliver ARM-Powered Supercomputer to UK Consortium. (2017). https://www.top500.org/news/cray-to-deliver-arm-powered-supercomputer-to-uk-consortium/. (cited on page 168)

Feng, W.-c.; Balaji, P.; Baron, C.; Bhuyan, L. N.; and Panda, D. K., 2005. Performance characterization of a 10-gigabit ethernet toe. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, 58–63. IEEE. (cited on page 111)

Frigo, M. and Johnson, S. G., 1998. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, 1381–1384. IEEE. (cited on page 40)

Fu, H.; Liao, J.; Yang, J.; Wang, L.; Song, Z.; Huang, X.; Yang, C.; Xue, W.; Liu, F.; Qiao, F.; et al., 2016. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59, 7 (2016), 1–16. (cited on pages 2, 167, and 169)

Fu, S.; Chang, R.; Couture, S.; Menarini, M.; Escobar, M.; Kuteifan, M.; Lubarda, M.; Gabay, D.; and Lomakin, V., 2015. Explore computational power of mobile platforms in micromagnetic simulations. (2015). (cited on page 40)

Fujitsu, 2016. Fujitsu's Next Endeavor: The Post-K Computer. (2016). https://www.fujitsu.com/global/Images/fujitsu-next-endeavor-the-post-k-computer.pdf. (cited on pages 3 and 168)

Garzón, E.; Moreno, J.; and Martínez, J., 2016. An approach to optimise the energy efficiency of iterative computation on integrated gpu-cpu systems. *The Journal of Supercomputing*, (2016), 1–12. (cited on page 43)

Ge, R.; Feng, X.; Burtscher, M.; and Zong, Z., 2014. Performance and energy modeling for cooperative hybrid computing. In *Networking, Architecture, and Storage (NAS), 2014 9th IEEE International Conference on*, 232–241. IEEE. (cited on page 46)

Gepner, P.; Gamayunov, V.; and Fraser, D., 2011. Early performance evaluation of AVX for HPC. *Procedia Computer Science*, 4 (2011), 452–460. (cited on page 40)

GNU, 2017. Offloading Support in GCC. (2017). https://gcc.gnu.org/wiki/Offloading. (cited on page 86)

Gonzalez, R. C. and Woods, R. E., 2001. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. ISBN 0201180758. (cited on page 54)

González-Domínguez, J.; Schmidt, B.; Kässens, J. C.; and Wienbrandt, L., 2014. Hybrid CPU/GPU Acceleration of Detection of 2-SNP Epistatic Interactions in GWAS. In *Euro-Par 2014 Parallel Processing*, 680–691. Springer. (cited on page 44)

Grasso, I.; Radojkovic, P.; Rajovic, N.; Gelado, I.; and Ramirez, A., 2014. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 123–132. IEEE. (cited on pages 3 and 39)

Green500. Green 500 List. http://www.green500.org/. Accessed: 29/06/2017. (cited on pages 3 and 4)

Gschwind, M., 2012. Blue Gene/Q: design for sustained multi-petaflop computing. In *Proceedings of the 26th ACM international conference on Supercomputing*, 245–246. ACM. (cited on page 26)

Han, T. D. and Abdelrahman, T. S., 2009. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on*

General Purpose Processing on Graphics Processing Units, 52–61. ACM. (cited on page 42)

Heinecke, A.; Klemm, M.; Pabst, H.; and Pflüger, D., 2012. Towards high-performance implementations of a custom HPC kernel using® array building blocks. *Facing the Multicore-Challenge II*, (2012), 36–47. (cited on page 41)

Heinecke, A. and Pflüger, D., 2011. Multi-and many-core data mining with adaptive sparse grids. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 29. ACM. (cited on page 40)

Hennessy, J. L. and Patterson, D. A., 2003. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1558607242. (cited on page 14)

Hoeflinger, J. P. and de Supinski, B. R., 2005. The openmp memory model. In *IWOMP*, vol. 4315 of *Lecture Notes in Computer Science*, 167–177. Springer. (cited on page 33)

HP, 2014. HP moonshot system. http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/index.aspx. (cited on page 17)

HPE, 2013. HPE Proliant m800 Cartridge. (2013). (cited on page 24)

HPE, 2017. HPE Moonshot Family. (2017). (cited on page 23)

Ibrahim, M.; Rupp, M.; and Fahmy, H., 2009. Code transformations and SIMD impact on embedded software energy/power consumption. In *Computer Engineering & Systems, 2009. ICCES 2009. International Conference on*, 27–32. IEEE. (cited on page 26)

Igual, F. D.; Ali, M.; Friedmann, A.; Stotzer, E.; Wentz, T.; and van de Geijn, R. A., 2012. Unleashing the high-performance and low-power of multi-core dsps for general-purpose hpc. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 26. IEEE Computer Society Press. (cited on pages 4, 41, 75, 76, 85, 96, 129, 181, and 182)

Intel, 2011. Intel 64 and IA-32 Architectures Software DeveloperâĂŹs Manual, Volume 3A, 3B and 3C: System Programming Guide, Parts 1 and 2. (2011). (cited on page 45)

Intel Corporation, 2007. Intel C++ Intrinsic Reference. (2007). Accessed: 29/11/2012. (cited on pages 27 and 28)

Intel Corporation, 2010. Intel Atom™ Processors N450 , D410 and D510 for Embedded Computing. (2010). Accessed: 14/10/2012. (cited on page 56)

Intel Corporation, 2012. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes*. (cited on page 70)

Intel Corporation, 2014. Intel core m-5y processor series. http://www.intel.com.au/content/www/au/en/processors/core/core-m-processors.html. (cited on page 2)

Jeffers, J.; Reinders, J.; and Sodani, A., 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann. (cited on page 167)

Jeong, H.; Kim, S.; Lee, W.; and Myung, S.-H., 2012. Performance of SSE and AVX Instruction Sets. *arXiv preprint arXiv:1211.0820*, (2012). (cited on page 40)

Jeun, W.-C. and Ha, S., 2007. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, 44–49. IEEE Computer Society. (cited on page 41)

Jones, D. L., 2010. The μCurrent. http://alternatezone.com/electronics/ucurrent/uCurrentArticle.pdf. (cited on page 138)

Keipert, K.; Mitra, G.; Sunriyal, V.; Leang, S. S.; Sosonkina, M.; Rendell, A. P.; and Gordon, M. S., 2015. Energy-efficient computational chemistry: Comparison of x86 and arm systems. *Journal of Chemical Theory and Computation*, 11, 11 (2015), 5055–5061. doi:10.1021/acs.jctc.5b00713. http://dx.doi.org/10.1021/acs.jctc.5b00713. PMID: 26574303. (cited on page 172)

Khronos, 2011. OpenCL: The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl. (cited on pages 37 and 84)

Kirk, D., 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *International Symposium on Memory Management: Proceedings of the 6 th international symposium on Memory management*, vol. 21, 103–104. (cited on page 26)

Komoda, T.; Hayashi, S.; Nakada, T.; Miwa, S.; and Nakamura, H., 2013. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 349–356. IEEE. (cited on page 46)

Kutil, R., 2008. Parallelization of IIR filters using SIMD extensions. *2008 15th International Conference on Systems, Signals and Image Processing*, 1, June (Jun. 2008), 65–68. doi:10.1109/IWSSIP.2008.4604368. (cited on page 40)

LaKomski, D.; Zong, Z.; Jin, T.; and Ge, R., 2015. Optimal balance between energy and performance in hybrid computing applications. In *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*, 1–8. IEEE. (cited on page 43)

Lamport, L., 1974. The parallel execution of do loops. *Commun. ACM*, 17, 2 (1974), 83–93. doi:http://doi.acm.org/10.1145/360827.360844. (cited on page 35)

Landaverde, R.; Zhang, T.; Coskun, A. K.; and Herbordt, M., 2014. An Investigation of Unified Memory Access Performance in CUDA. (2014). (cited on page 39)

Lang, J. and Rünger, G., 2014. An execution time and energy model for an energy-aware execution of a conjugate gradient method with cpu/gpu collaboration. *Journal of Parallel and Distributed Computing*, 74, 9 (2014), 2884–2897. (cited on page 46)

Leang, S. S.; Rendell, A. P.; and Gordon, M. S., 2014a. Quantum Chemical Calculations Using Accelerators: Migrating Matrix Operations to the NVIDIA Kepler GPU and the Intel Xeon Phi. *Journal of Chemical Theory and Computation*, 10, 3 (2014), 908–912. (cited on page 42)

Leang, S. S.; Rendell, A. P.; and Gordon, M. S., 2014b. Quantum Chemical Calculations Using Accelerators: Migrating Matrix Operations to the NVIDIA Kepler GPU and the Intel Xeon Phi. *Journal of Chemical Theory and Computation*, 10, 3 (2014), 908–912. doi:10.1021/ct4010596. http://pubs.acs.org/doi/abs/10.1021/ct4010596. (cited on page 42)

Lebacki, B.; Wolfe, M.; and Miles, D., 2012. The PGI Fortran and C99 OpenACC Compilers. *Cray User Group*, (2012). (cited on page 42)

Lee, S.; Meredith, J. S.; and Vetter, J. S., 2015. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th*

ACM on International Conference on Supercomputing, 405–414. ACM. (cited on page 43)

Liao, C.; Hernandez, O.; Chapman, B.; WenguangChen; and Zheng, W., 2006. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience, Special Issueon CPC'2006 selected papers*, (2006). (cited on page 32)

Liao, C.; Yan, Y.; de Supinski, B. R.; Quinlan, D. J.; and Chapman, B., 2013. Early Experiences With The OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators*, 84–98. Springer. (cited on page 42)

Lin, Z., 2011. Delivering performance, efficiency and differentiation with tiâĂŹs multistandard base station socs. *Texas Instruments White Paper*, (2011). (cited on page 18)

Linderman, M. D.; Collins, J. D.; Wang, H.; and Meng, T. H., 2008. Merge: a programming model for heterogeneous multi-core systems. In *ACM SIGOPS operating systems review*, vol. 42, 287–296. ACM. (cited on page 44)

Maleki, S.; Gao, Y.; Garzarán, M.; Wong, T.; and Padua, D., 2011. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 372–382. IEEE. (cited on page 41)

Maruyama, T.; Yoshida, T.; Kan, R.; Yamazaki, I.; Yamamura, S.; Takahashi, N.; Hondou, M.; and Okano, H., 2010. SPARC64 VIIIFX: A New-Generation Octocore Processor for Petascale Computing. *Micro, IEEE*, 30, 2 (2010), 30–40. (cited on page 26)

McVoy, L.; Staelin, C.; et al., 1996. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 279–294. San Diego, CA, USA. (cited on page 50)

McVoy, L. W. and Staelin, C., 1996. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 279–294. citeseer.ist.psu.edu/mcvoy96lmbench.html. (cited on page 52)

Mitra, G.; Johnston, B.; Rendell, A. P.; McCreath, E.; and Zhou, J., 2013. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE. (cited on page 168)

nCore HPC, 2014. ncore browndwarf y-class supercomputer. http://ncorehpc.com/browndwarf/. (cited on page 17)

Newburn, C.; Dmitriev, S.; Narayanaswamy, R.; Wiegert, J.; Murty, R.; Chinchilla, F.; Deodhar, R.; and McGuire, R., 2013. Offload Compiler Runtime for the Intel Xeon Phi Coprocessor. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, 1213–1225. (cited on page 42)

Note, F. W.; Van Zee, F. G.; Smith, T.; Igual, F. D.; Smelyanskiy, M.; Zhang, X.; Kistler, M.; Austel, V.; Gunnels, J.; Low, T. M.; et al., 2013. Implementing level-3 blas with blis: Early experience. (2013). (cited on pages 41 and 181)

NVIDIA, 2012. NVML API REFERENCE MANUAL. (2012). (cited on page 45)

NVIDIA, 2014a. NVIDIA Tegra K1 Processor. http://www.nvidia.com/object/tegra-k1-processor.html. (cited on page 42)

NVIDIA, 2014b. Unified Memory in CUDA 6. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. (cited on page 42)

Ohshima, S.; Kise, K.; Katagiri, T.; and Yuba, T., 2007. Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment. In *High Performance Computing for Computational Science-VECPAR 2006*, 305–318. Springer. (cited on page 43)

OpenMP, 2007. OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org. (cited on page 27)

OpenMP ARB, 2015. OpenMP Application Program Interface, v.4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (cited on page 169)

Papadrakakis, M.; Stavroulakis, G.; and Karatarakis, A., 2011. A New Era in Scientific Computing: Domain Decomposition Methods in Hybrid CPU-GPU Architectures. *Computer Methods in Applied Mechanics and Engineering*, 200, 13 (2011), 1490–1508. (cited on page 44)

Pulli, K.; Baksheev, A.; Kornyakov, K.; and Eruhimov, V., 2012. Real-time computer vision with OpenCV. *Communications of the ACM*, 55, 6 (2012), 61–69. (cited on page 41)

Rajovic, N.; Carpenter, P. M.; Gelado, I.; Puzovic, N.; Ramirez, A.; and Valero, M., 2013a. Supercomputing with commodity cpus: are mobile socs

ready for hpc? In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 40. ACM. (cited on pages 2 and 38)

Rajovic, N.; Rico, A.; Puzovic, N.; Adeniyi-Jones, C.; and Ramirez, A., 2013b. Tibidabo: Making the case for an arm-based hpc system. *Future Generation Computer Systems*, (2013). (cited on page 17)

Rajovic, N.; Rico, A.; Puzovic, N.; Adeniyi-Jones, C.; and Ramirez, A., 2014. Tibidabo: Making the Case for an ARM-based HPC System. *Future Generation Computer Systems*, 36 (2014), 322–334. (cited on pages 39 and 45)

Rajovic, N.; Vilanova, L.; Villavieja, C.; Puzovic, N.; and Ramirez, A., 2013c. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4, 6 (2013), 439–443. (cited on page 3)

Ren, D.-Q. and Suda, R., 2012. Global optimization model on power efficiency of gpu and multicore processing element for simd computing with cuda. *Computer Science-Research and Development*, 27, 4 (2012), 319–327. (cited on page 43)

Reyes, R.; Lopez, I.; Fumero, J. J.; and de Sande, F., 2012. Directive-based Programming for GPUs: A Comparative Study. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE 14th International Conference on*, 410–417. IEEE. (cited on page 42)

Rintaluoma, T., 2009. Optimizing H. 264 Decoder for Cortex-A8 with ARM NEON OpenMax DL Implementation. *IQ Magazine: The Smart Approach to Designing with the ARM Architecture*, 8, 2 (2009), 32–37. (cited on page 40)

Rintaluoma, T. and Silven, O., 2010. SIMD performance in software based mobile video coding. *Embedded Computer System*, (Jul. 2010), 79–85. (cited on page 40)

Sao, P.; Vuduc, R.; and Li, X. S., 2014. A Distributed CPU-GPU Sparse Direct Solver. In *Euro-Par 2014 Parallel Processing*, 487–498. Springer. (cited on page 44)

Schmidl, D.; Cramer, T.; Wienke, S.; Terboven, C.; and Müller, M. S., 2013. Assessing the performance of OpenMP programs on the Intel Xeon Phi. In *Euro-Par 2013 Parallel Processing*, 547–558. Springer. (cited on page 42)

Shalf, J.; Dosanjh, S.; and Morrison, J., 2011. Exascale computing technology challenges. *High Performance Computing for Computational Science–VECPAR 2010*, (2011), 1–25. (cited on pages 1 and 170)

Shin, F., 2010. *Image Processing and Pattern Recognition: Fundamentals and Techniques*. John Wiley & Sons. (cited on page 176)

Smith, T. M.; van de Geijn, R. A.; Smelyanskiy, M.; Hammond, J. R.; and Van Zee, F. G., 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*. (cited on pages 120 and 150)

Stephens, N.; Biles, S.; Boettcher, M.; Eapen, J.; Eyole, M.; Gabrielli, G.; Horsnell, M.; Magklis, G.; Martinez, A.; Premillieu, N.; et al., 2017. The arm scalable vector extension. *IEEE Micro*, 37, 2 (2017), 26–39. (cited on page 168)

Texas Instruments, 2011. DM3730, DM3725 Digital Media Processors. Technical Report July. (cited on page 56)

Texas Instruments Inc., 2010a. Texas Instruments C66x DSP Corepac. (2010). (cited on pages xv and 13)

Texas Instruments Inc., 2010b. Texas Instruments Keystone I C6678. (2010). (cited on pages xv, 14, and 15)

Texas Instruments Inc., 2012. Texas Instruments Keystone II 6638K2H. (2012). (cited on pages xv, 15, and 16)

Tiwari, A.; Jundt, A.; Ward, W. A.; Campbell, R.; and Carrington, L., 2015a. Building blocks for a system-wide power and thermal management framework. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, 700–707. IEEE. (cited on page 45)

Tiwari, A.; Keipert, K.; Jundt, A.; Peraza, J.; Leang, S. S.; Laurenzano, M.; Gordon, M. S.; and Carrington, L., 2015b. Performance and energy efficiency analysis of 64-bit arm using gamess. In *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, 8. ACM. (cited on page 46)

Tiwari, A.; Laurenzano, M.; Carrington, L.; and Snavely, A., 2012a. Auto-tuning for Energy Usage in Scientific Applications. In *Euro-Par 2011: Parallel Processing Workshops*, vol. 7156 of *Lecture Notes in Computer Science*,

178–187. Springer Berlin Heidelberg. ISBN 978-3-642-29739-7. doi:10.1007/978-3-642-29740-3_21. (cited on page 45)

Tiwari, A.; Laurenzano, M. A.; Carrington, L.; and Snavely, A., 2012b. Modeling power and energy usage of hpc kernels. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, 990–998. IEEE. (cited on pages 45 and 46)

Top500. Top 500 List. http://www.top500.org/. Accessed: 29/06/2017. (cited on page 2)

Turnes, C. K. and Romberg, J., 2010. Spiral fft: an efficient method for 3-d ffts on spiral mri contours. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, 617–620. IEEE. (cited on page 40)

Varghese, A.; Edwards, B.; Mitra, G.; and Rendell, A. P., 2014. Programming the adapteva epiphany 64-core network-on-chip coprocessor. *CoRR*, abs/1410.8772 (2014). http://arxiv.org/abs/1410.8772. (cited on page 39)

Wan, J.; Wang, R.; Lv, H.; Zhang, L.; Wang, W.; Gu, C.; Zheng, Q.; and Gao, W., 2012. AVS video decoding acceleration on ARM Cortex-A with NEON. In *Signal Processing, Communication and Computing (ICSPCC), 2012 IEEE International Conference on*, 290–294. IEEE. (cited on page 40)

Whaley, R. C. and Dongarra, J. J., 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, 1–27. IEEE Computer Society. (cited on pages 122 and 181)

Wienke, S.; Springer, P.; Terboven, C.; and an Mey, D., 2012. OpenaccâĂŤfirst experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, 859–870. Springer. (cited on page 169)

Wolfe, M., 2010. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 43–50. ACM. (cited on page 42)

Wulf, W. A. and McKee, S. A., 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23, 1 (1995), 20–24. (cited on page 25)

Yang, C.; Wang, F.; Du, Y.; Chen, J.; Liu, J.; Yi, H.; and Lu, K., 2010. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, 19–28. IEEE. (cited on pages 43 and 44)

ZHANG, J.; SU, H.-B.; AND WU, Q.-Z., 2009. Research and implement of serial rapidio based on mul-dsp. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 1–4. IEEE. (cited on page 17)