HIGH-LEVEL PROGRAMMING MODEL FOR HETEROGENEOUS EMBEDDED SYSTEMS USING MULTICORE INDUSTRY STANDARD APIS

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Peng Sun

August 2016

HIGH-LEVEL PROGRAMMING MODEL FOR HETEROGENEOUS EMBEDDED SYSTEMS USING MULTICORE INDUSTRY STANDARD APIS

Peng Sun

APPROVED:

Dr. Chapman, Barbara
Dept. of Computer Science, University of Houston

Dr. Gabriel, Edgar
Dept. of Computer Science, University of Houston

Dr. Shah, Shishir
Dept. of Computer Science, University of Houston

Dr. Subhlok, Jaspal
Dept. of Computer Science, University of Houston

Dr. Chandrasekaran, Sunita
Dept. of CIS, University of Delaware

Dean, College of Natural Sciences and Mathematics

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Barbara Chapman, for her invaluable advice and guidance in my Ph.D. study. I appreciate all her dedicated guidance, and great opportunities to participate in those worthwhile academic projects, and the funding to complete my Ph.D. study. Her passion and excellence on academic and contributions to communities encourage me to finish my Ph.D. degree.

Specifically, I am very grateful to Dr. Sunita Chandrasekaran for the mentoring and guidance for my research and publications. That help is paramount to my Ph.D. Journey. I truly could not achieve the degree without her mentoring and advisory.

Special thanks to all my committee members: Dr. Edgar Gabriel, Dr. Shishir Shah, Dr. Jaspal Subhlok, for their time, insightful comments and help.

I would also like to thank my fellow labmates of the HPCTools group: Dr. Deepak Eachempati, Tony Curtis, Dr. Dounia Khaldi, Dr. Cheng Wang, Dr. Xiaonan Tian for their valuable discussion and friendships. I enjoy working with them and discussions particularly in the weekly meetings.

During my five years intensive and productive Ph.D. journey, I also have the extreme fortune to have opportunities to have internships with Freescale and AMD. For my two terms of internships in Freescale, I work closely with John Russo, Fred Peterson, and Udi Kalekin; I learned a lot from their excellence and in-depth knowledge of compilers and system architectures. Also huge thanks to Adrian Edwards and Ben Sander from AMD, for their patience and insightful guidance and generous support. I would like to extend my sincere appreciation them all.

Last, but not least, my dissertation is dedicated to my wife, Xueming Zhou, my

son, Luffy Sun, and Loren Sun, my second boy to cuddle in two months. Also, I cannot express my appreciation for endless supports from my parents, Wenchang Sun and Xiurong Li, my father and mother in law, Rongqiang Zhou, and Ting Zhang. Their love and selflessly support provided me the driving force, courage, and perseverance. Thank you all for supporting me along the way.

HIGH-LEVEL PROGRAMMING MODEL FOR HETEROGENEOUS EMBEDDED SYSTEMS USING MULTICORE INDUSTRY STANDARD APIS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Peng Sun

August 2016

Abstract

Multicore embedded systems are rapidly emerging. Hardware designers are packing more and more features into their design. Introducing heterogeneity in these systems, i.e., adding cores of varying types, provides opportunities to solve problems in different aspects. However, those designs present several challenges to embedded system programmers since software is still not mature enough to efficiently exploit the capabilities of emerging hardware, gorgeous with cores of varying architectures.

Programmers still rely on understanding and using low-level hardware-specific APIs. The approach is not only time-consuming but also tedious and error-prone. Moreover, the solutions developed are very closely tied to a particular hardware, raising significant concerns over software portability. What is needed is an industry standard that will enable better programming practices for both current and future embedded systems. To that end, in this dissertation, we have explored the possibility of using existing standards, such as OpenMP, that provide portable high-level programming constructs along with the industry-driven standards for multicore systems. We built a portable yet lightweight OpenMP runtime library that incorporates the Multicore Association APIs, making OpenMP programming model available to embedded-system programmers with a broad coverage of targeting embedded devices. In this dissertation, we also explore how to use industry standard APIs as the mapping layer of OpenMP onto heterogeneous-embedded systems. By adopting HIP as the plugin to our stack, we could portably map the applications to heterogeneous devices from different vendors with single-code space.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce the motivation of the dissertation topic, the main contribution, and the outlines of this dissertation report.

1.1 Motivation

Programming embedded systems is a challenge, the use of low-level proprietary Application Programming Interfaces (APIs) makes the development process tedious and error-prone. Embedded systems are becoming more heterogeneous, for better performance and power efficiency. Hardware designers pack more and more specialized processors into the new embedded systems and System-on-Chips(SoCs) such as DSPs, FPGAs, and GPUs. Those hardware elements have different architectures and instruction sets, making software development more complicated. Thus a steep learning curve follows.

Furthermore, programming tools that could help programmers efficiently oversee and utilize the computation power of these hardware remains a challenge, especially considering the development of high-performance parallel applications. Developers will need to tackle the distribution of computations and tasks among the different computation elements available, with explicit synchronization and communication. Without a portable and easily adopted programming model, the efficient use of that hardware is not possible.

OpenMP [22] is considered a potential candidate to resolve most challenges we discussed above. OpenMP is a pragma-based programming model; it allows the software developers to parallelize sequential code incrementally, providing data parallelism or task parallelism without the need to handle lower-level threading libraries or Operating-System-level (OS-level) resources. Besides, OpenMP allows a compiler to ignore the directives thus further guarantees the portability of software products. With the release of OpenMP 4.5 specification [6], OpenMP expanded its coverage to heterogeneous architectures. The OpenMP *target* construct allows programs to dispatch part of the executions onto accelerators. Early studies that implement and explore the new OpenMP features include [58, 42].

There is still significant impediments to map OpenMP onto the embedded domain despite the high potential for using OpenMP on heterogeneous multicore-embedded systems. Unlike commonly guaranteed features in the general-purpose computation devices, embedded systems are typically short of hardware resources or OS-level support. As an example, most OpenMP compilers translate the pragmas into parallel code with OS-level threading libraries. Cache coherence is expected by the OpenMP

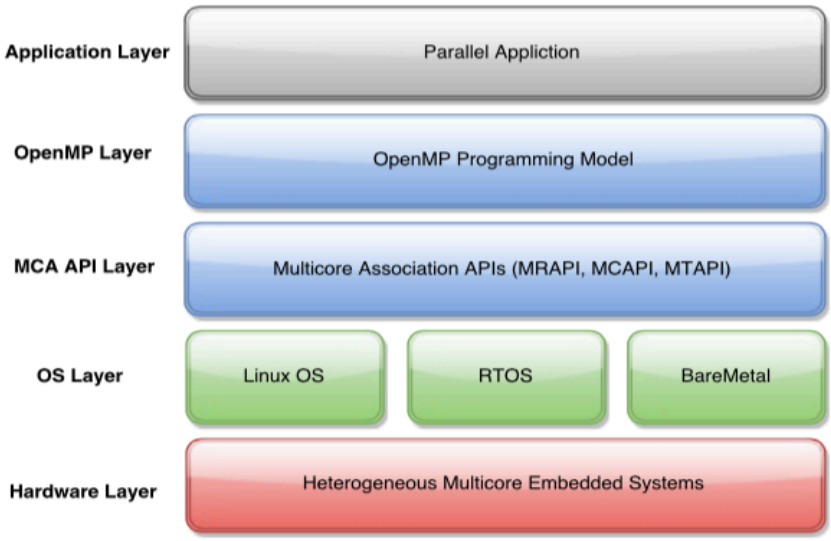

Figure 1.1: Solution stack using OpenMP on MCA APIs

compilers. Not surprisingly, embedded systems lack some of these features such as OS.

Figure 1.1 illustrates the solution stack to resolve the challenges discuss above. We choose to deploy the APIs defined by the **M**ulti**C**ore **A**ssociation (MCA) [4] as the translation layer of the OpenMP runtime library. MCA is a non-profit industry consortium formed by a group of leading semiconductor companies and academics. It defines open standards for multicore embedded systems. There are three major specifications established by the MCA, the Resource Management API (MRAPI), the Communication API (MCAPI), and the Task Management API (MTAPI). We have explored mapping of OpenMP runtime on MRAPI and the suitability of using MCAPI across embedded nodes in our previous work [68, 60]. The results from our past work demonstrated that such an approach, translating OpenMP to MCA APIs,

has potential for both programmability and portability.

In this dissertation, we propose a comprehensive-solution stack and expandable platform, to help programmers explore parallelism on heterogeneous-embedded platforms with the support of OpenMP. Our stack uses MCA APIs as the mapping layer, with suitable plug-ins and an optimized scheduler, to provide a portable OpenMP implementation across heterogeneous-embedded platforms.

1.2 Contributions

The main contributions of this dissertation are:

- Explore the suitability of mapping OpenMP onto embedded systems with MRAPI, MTAPI, and other industry standards.
- Extend MCA Task Management API to support heterogeneous systems across vendors.
- Adopt MCA APIs to represent lower-level system resources, providing OpenMP-sufficient parallel-execution support.

1.3 Dissertation Outline

This section gives an introduction of the outlined introduction of the dissertation. In Chapter 2 we review the background of the dissertation related technologies and

concepts, which includes a high-level programming model, and the low-level industry standards that abstract the underlying system, and software details. Chapter 3 discusses the related work of the dissertation, the available work regarding programming parallel applications on embedded systems, and the existing effort to use industry standards for abstraction. Chapter 4 describes our investigation and experiments that choose the proper plugin language for our solution stack and offloading the computation tasks onto heterogeneous platforms. Chapter 5 provides the work we have done towards MCA, the industry-standard APIs; including the efforts for each of the MCA APIs towards a broader support target platforms, an extension to support on heterogeneous systems, and a discussion on performance. Chapter 6 discusses the mapping of OpenMP, the higher-level programming model, onto MCA APIs, to provide a convenient programming model while guaranteeing the software portability. Chapter 7 discusses the overall contribution and the future work.

Chapter 2

Background

In this chapter, we introduce the background of the dissertation. In this work, we adopt OpenMP as the programming model, with the help of MCA APIs and HIP, to map on various homogeneous and heterogeneous systems. We also highlight key features and usability, which will better help understand our dissertation and the solution stack to the challenges. Noted the background of **H**eterogeneous-programming **I**nterface for **P**ortability (HIP) and **H**eterogeneous **S**ystem **A**rchitectures (HSA) will be introduced in Section 5.3.

2.1 OpenMP

OpenMP [22] is a well-known portable and scalable programming model that facilitates programmers with simple, but a versatile interface to develop parallel applications in both homogeneous or heterogeneous environments. By inserting pragmas

into a regular program, OpenMP makes it easy for programmers to leverage the underlying hardware rapidly and effortlessly. The primary model of OpenMP for parallel execution is fork - join.

OpenMP has the feature that allows programmers to gradually parallelize sequential codes. This makes OpenMP the best choice when developers want to explore the potential parallelism of existing-legacy programs. We use MCA Resource Management APIs to manage the system thread-level resource and implement OpenMP data-parallelization functionality. Besides, we also map OpenMP task construct onto MCA Task Management APIs. This provides programmers with an easy-to-use programming interface for task parallelism for broad coverage of embedded systems. Therefore, we will introduce the related OpenMP concepts in this section.

```
1 void sum(int n, float *a, float *b)
2 {
3  int i;
4 #pragma omp parallel for
5  for (i=1; i<n; i++)
6  b[i] = (a[i] + a[i-1]) / 2.0;
7 }
```

Listing 2.1: OpenMP example

As seen in the code snippet in Listing 2.1, the program begins with a single thread execution, when encounters the parallel region marked as *#pragma omp parallel for*, the master thread will create or fork a team of worker threads to compute the workloads in parallel. After the computation, all the worker threads will synchronize and join, leaving only the master thread to continue for the rest of computations.

OpenMP *task* and *taskwait* constructs were introduced in OpenMP 3.0 specification. They are updated with the definition of *taskgroup* and task dependency clauses in OpenMP 4.0 specification. OpenMP defines a task as a particular instance of execution that contains the executable codes and its data frame. The task constructs can be placed anywhere inside an OpenMP *parallel* construct, and explicit tasks are created when a thread encounters the *task* construct. Then these tasks can be synchronized by using the *taskwait* construct. The tasks must pause and wait for their child tasks to be completed at the *taskwait* form cause before moving on to the next stage in the program.

```
int fib(int n)
{
  int i, j;
  if (n<2) return n;
  else{
       #pragma omp task shared(i)
       i=fib(n-1);
       #pragma omp task shared(j)
       j=fib(n-2);
       #pragma omp taskwait
       return i+j;
  }
}
int main(void){
  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

Listing 2.2: OpenMP task example

The code snippet in Listing 2.2 illustrates the implementation of OpenMP task parallelism solution of Fibonacci numbers.

Although programmers targeting general-purpose computation facilities have conveniently enjoyed parallelizing serial codes using OpenMP for many years, most of the embedded-system programmers have not been able to exploit this functionality. Typical embedded systems lack features found on general-purpose computers that are essential for OpenMP execution. OpenMP implementations usually rely heavily on threading libraries and OS-level resources. However, a typical embedded system

may lack such features. Also, some embedded systems may not have an OS such as a bare-metal device. It might seem simple to offload the most compute-intensive portions of the code to an accelerator using conventional methods, such as remote procedure calls. However, alternate dataflow-oriented approaches have proved to be more efficient due to lesser application deadlocks, for devices such as FPGAs [12]. This hurdle requires programmers to re-think what kind of specialized resources an embedded device can offer.

It remains a challenge to create an efficient (almost universal) programming interface for the embedded world. Because such platforms can not provide substantial details of the hardware or efficient tools for application optimization.

2.2 MCA Libraries

To address some of the fundamental issues of programming multicore-embedded systems, the Multicore Association (MCA) was founded by a group of leading semiconductor companies, embedded-solution companies, and academics, aiming to form industry standards for programming embedded systems.

The Multicore Association API [4] defined *MRAPI* (the multicore resource management API), *MCAPI* (the multicore communication API) and *MTAPI* (the multicore task management API), as the fundamental stands. With the three sets of API designed exclusively for the embedded systems, MCA APIs aim to abstract the lower-level hardware details and provide a unique API interface across platforms, to rapidly implement the software development and quick exploration of parallelism

and heterogeneous computations. We will introduce the three major APIs from the Multicore Association in the following paragraphs.

2.2.1 MCA Resource Management APIs

MRAPI handles resource-management challenges of the most critical hardware resources on real products, including shared memory, remote memory, synchronization primitives, and metadata, for both SMP and AMP architectures. MRAPI can support any number of cores, even each with a different instruction set, same or different OSes. Besides that, MRAPI allows coordinated concurrent access to the systems resources by deploying the synchronization primitives for embedded systems that with limited hardware resources. The MRAPI supports a variety of operating systems, including Embedded Linux, RTOS, and even Bare-Metal systems. We would like to summarize some of the fundamental concepts of MRAPI since we will be using this functionality in our software design.

2.2.1.1 Domain and Nodes

The MRAPI systems are composed of one or more *domain*; each domain is considered as a unique system global entity. An MRAPI *node* is an independent unit of execution, and an MRAPI domain will comprise a team of MRAPI nodes. Each node can map to any execution unit such as a process, thread, a thread-pool, or a hardware accelerator.

2.2.1.2 Memory Primitives

With the concept of heterogeneity in mind, MRAPI supports two different memory models, the shared memory, and the remote memory. Shared memory primitives allow users to manage the on-chip or off-chip shared memory directly, with assigned attributes. Unlike the Linux shared memory, which is only accessible within one operating system's entity, the MRAPI shared memory can be accessed by different nodes running different OSes. The remote memory model enables the access of distinct memories. This model can be physically consecutive or not direct access, for the latter case, some other methods like DMA can be used to access the remote memory. By providing unique API interfaces, MRAPI hides all memory access details from the end users.

2.2.1.3 Synchronization Primitives

MRAPI offers a set of synchronization primitives including *Mutexes*, *Semaphores* and *Reader/Writer locks*. These synchronization primitives guarantee the MRAPI nodes properly access the shared resources, to avert data race or race conditions.

2.2.1.4 System Resource Metadata

MRAPI specification provides a facility for retrieval of system metadata in a resource tree format, providing details of resources availability for the target system.

MRAPI is an excellent candidate that can simplify the interface of underlying

hardware and OS-level resources, for programmers as well as the higher-level language such as OpenMP. In the dissertation, we propose extension and implementation of MRAPI with thread-level resource management and use the extended MRAPI implementation to be the mapping layer of our OpenMP implementation. Performance evaluations in the later chapter will prove that by adding MRAPI as an abstract layer, our OpenMP-MCA runtime library (RTL) will not incur extra overheads and achieve compatible performance.

2.2.2 MCA Communication APIs

MCAPI is designed to capture the core elements of communication and synchronization required for closely distributed embedded systems, as a message-passing API. Industry vendors such as [7, 5] have also provided MCAPI support for their products.

The purpose of MCAPI, which is a message-passing API, is to capture the essential elements of communication and synchronization that are required for closely distributed embedded systems. MCAPI provides a limited number of calls with sufficient communication functionalities while keeping it simple enough to allow efficient implementations. Additional functionality can be layered on top of the API set. The calls are exemplifying functionality and are not mapped to any particular existing implementation.

MCAPI defines three types of communication:

- Messages, connectionless diagrams.

- Package Channel, connection-oriented, uni-directional, FIFO package streams.
- Scalar Channel, connection-oriented, single-word uni-directional, FIFO package streams.

MCAPI messages provide a flexible method to transmit data between endpoints without first establishing a connection. The buffers on both sender and receiver sides must be furnished by the user application. MCAPI messages may be sent with different priorities. MCAPI packet channels provide a method to transmit data between endpoints by first establishing a connection, thus potentially removing the message header and route discovery overhead. Packet channels are unidirectional and deliver data in a FIFO (first in first out) manner. The buffers are provided by the MCAPI implementation on the receive side, and by the user application on the send side. MCAPI scalar channels provide a method to transmit scalars very efficiently between endpoints by first establishing a connection. Like packet channels, scalar channels are unidirectional and deliver data in a FIFO (first in first out) manner. The scalar functions come in 8-bit, 16-bit, 32-bit, and 64-bit variants. The scalar receives must be of the same size as the scalar sends. A mismatch in size results in error.

In the work discussed in Chapter 5, we based on the reference MCAPI implementation from the MCA official website. We deployed all their API implementation and modified the transportation-layer implementation to target our platform specifically.

2.2.3 MCA Task Management APIs

In this section, we briefly introduce OpenMP and MCA Task Management API (MTAPI) by discussing their essential features, usability, and suitability for the dissertation. The MCA Tasking API is aiming to manage the task-level parallelization of multicore embedded platforms. It includes complete support of task life-cycle, with optimization of task synchronization, scheduling, and load balancing. Siemens recently released its open-source MTAPI implementation [8] as a part of the EMBB [2] library.

The Multicore Association (MCA) is formed by a group of leading semiconductor companies and academies. Several functionality for the Multicore Association API [4] includes *MRAPI* (the multicore resource management API), *MCAPI* (the multicore communication API), and *MTAPI* (the multicore task management API). MCA APIs aim to abstract the lower-level hardware details and provide a unique and easy-to-use API for different embedded architectures, thus expedite the software development procedures.

To address the concerns of managing task parallelisms of embedded applications, MCA defined the Multicore Task API (MTAPI). MTAPI is designed to support both SMP and AMP systems, allowing an unlimited number of cores in the same or different architectures. Moreover, it allows minimum implementation on top of an embedded operating systems or even on bare metal with very limited hardware resources. MTAPI can ease the path to conducting portable task parallelisms on embedded devices. The central concepts of MTAPI are listed as follows:

2.2.3.1 Domain and Node

MTAPI system is comprised of one or more MTAPI domains. An MTAPI domain is a unique system global entity. Each MTAPI domain contains a set of MTAPI nodes. An MTAPI node is an independent unit of execution. A node can be a process, thread, a thread pool, a processor, a hardware accelerator, or an instance of an operating system. The mapping of MTAPI node is implementation-defined. In the MTAPI implementation, we use for this work, an MTAPI node compromises a team of worker threads.

2.2.3.2 Job and Action

A job is an abstraction of the work to be done. An action is the implementation of a particular job. Different actions can implement the same job. An action can be a software action or a hardware action. For instance, a job can be implemented by one action on a DSP board and another action on a general-purpose processor. The MTAPI API will pick a suitable action at runtime when executing the job.

2.2.3.3 TASK

An MTAPI task consists of a piece of code together with the data to be processed. A task is light-weight with fine granularity; thus, an application can create a large number of tasks. The tasks created within a node can be scheduled across different nodes, internally. Each task is attached to a particular action at the runtime. The tasks are scheduled by the MTAPI runtime scheduler. The MTAPI runtime library,

therefore, seeks an optimized way and a suitable action to handle the execution of the task. In this project, MTAPI tasks are created to address the OpenMP explicit tasks.

2.2.3.4 QUEUE

The queue is introduced to guarantee the sequential order of task execution. The tasks associated with the same queue object must be executed in the order that they are attached to the queue. This feature is useful in the scenarios where the data must be processed in a specified order.

2.2.3.5 TASK GROUP

Task groups allow synchronization of a group of tasks. Tasks attached to the same group must be completed before the next step by calling *groupwait*.

Primarily, the MTAPI specification is designed for embedded systems. Unlike other parallelism-programming models for the general-purpose computing systems, the MTAPI specification can be potentially developed for resource-limited devices, heterogeneous systems which consist of different computation units with different ISAs, devices using embedded OSes or even bare metal. MTAPI features are designed for accommodating various embedded architectures. Secondly, MTAPI is currently under active development by many embedded-system vendors and university researchers, as we have discussed in the related work section. The MRAPI and

MCAPI working groups continue to work with the MTAPI working group to identify gaps and improve the MCA features for enabling efficient mapping of tasks to embedded platforms. However, MTAPI is still a low-level library. A higher-level programming model is desired to improve the programmers' productivity. Thus, it is necessary to introduce OpenMP to the parallel-application development for embedded systems, with the lower-end support provided by MTAPI APIs.

We see MTAPI has certain advantages to being the mapping layer of our OpenMP-MCA RTL for the embedded-systems domain. In Chapter 5, we discussed the work we have done towards map OpenMP task support on top of MTAPI runtime library which achieved a good performance.

2.3 HIP and ROCm Stack

GPU has been promoted as the frequently used accelerator for both embedded systems and high-performance computing facilities. GPU computation accelerators, achieve significant power efficiency over the CPU alone. The modern programming model for GPU makes it much easier to port applications to GPU than the other hardware accelerators such as DSP or FPGAs. In this section, we introduce the background of the HIP and **R**adeon **O**pen **C**ompute Platfor**m** (ROCm) from the semiconductor company of AMD. Those tools provide an easy-to-adopt programming model and the fully open-source software stack to programmers that target GPUs across vendors.

2.3.1 HIP Programming Model

Compute **U**nified **D**evice **A**rchitecture (CUDA) has been widely used for GPU programming, a significant portion of libraries and applications utilizing GPU were built with CUDA. However, CUDA is a proprietary programming model of NVidia, which caused a series of drawbacks, including closed-source toolchain and lack of application portability.

AMD recently announced its GPUOpen initiative that provides a comprehensive open-source software stack-targeting programming on GPU. HIP, as part of the GPUOpen initiative, was intended to be adopted for both new application development and to port the existing GPU programs developed by CUDA. HIP is a thin layer that directs the compilation and execution of the applications onto either a NVidia or an AMD path. By maintaining single-code space of C++ applications with HIP routines, the programs can execute across platforms. Thus, the portability is firmly guaranteed. Besides, HIP supports the latest C++ language features such as templates and lambdas.

As illustrated in Figure 2.1, programs developed by HIP can be compiled and executed on platforms powered by both NVidia GPU and AMD GPU cards. On the NVidia path, the HIP compiler will incur NVCC as the C++ compiler, and translate the HIP routines in the applications to CUDA in the header files. In this way, the application developed in HIP will have no performance lose when comparing to the applications drawn up with CUDA on NVidia platforms. On the AMD path, HIP compiler will trigger the HCC compiler as the C++ compiler for HIP applications,

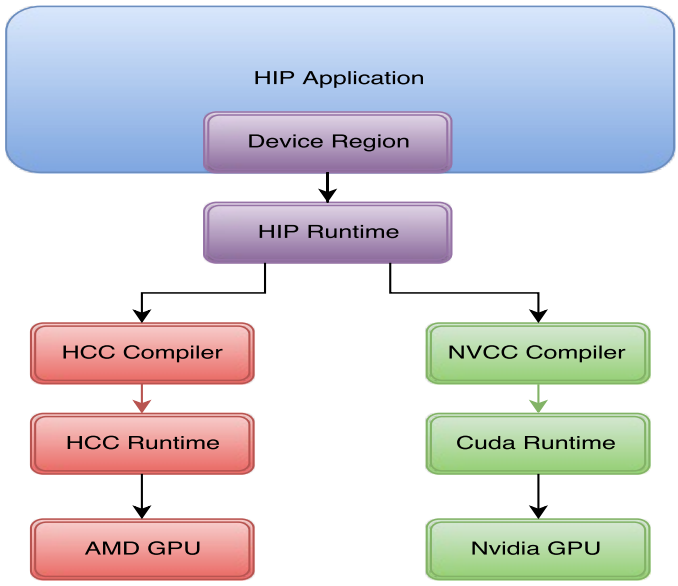

Figure 2.1: HIP on AMD and NV platforms

configure, and set the proper flags for compilation.

To aid the port of CUDA applications onto HIP, HIP provides a tool called "hipify," which performs a source-to-source translation over the existing CUDA applications onto HIP. The hipify tools are built with the C/C++ front-end of Low Level Virtual Machine (LLVM) compiler; thus, the hipify-tool code performs grammar analysis and makes sure the correctness of the translated CUDA applications.

2.3.2 ROCm Software Stack

ROCm, the open source software stack targeting AMD GPUs, and includes several components listed as follows:

- Linux Kernel Driver for GPU computing, available on both Debian and Fedora.
- User-mode runtime system based on HSA, for global-scale memory management and kernel dispatch.
- Heterogeneous C and C++ compiler that compiles both host code and device code offline.

The ROCm system is currently supported for the AMD Fiji GPUs, which features High-Bandwidth Memory(HBM). In our initial experiments on the ROCm stack, we observed large performance advantages of the Fiji GPUs over the NVidia GPUs on a set of data-streaming benchmarks, due to the extra-wide memory bandwidth provided by the HBM memory, of 512GB/sec.

With the release of ROCm by AMD, the GPU programmers were able to see the whole picture of GPU programming, from the top-layer programming model, runtime system, kernel driver, to the lower-level Instruction System Architecture(ISA) device code. Thus, the programmers have the privilege of exploring the full stack of the GPU apps, whether for debugging or performance tuning. This was not possible with the closed-source toolchain. In the next paragraphs, we will discuss the technologies used in ROCm stack, which are related to our decision on plugins for MTAPI.

2.3.2.0.1 ROCm Linux Driver The ROCm Linux driver initially targets AMD APUs that are fully compliant with the HSA specification. Those include Kaveri desktop APUs and Carrizo embedded/notebook APUs. It later extended to support of discrete GPU, which includes the FIJI series GPUs. The driver handles memory

allocation on both host and the GPU device, to provide interfaces for all stacks above in ROCm. It also includes multi-GPU system-level support with the shared view of host side memories for the GPUs in the system.

2.3.2.0.2 ROCK Runtime System ROCK Runtime System in ROCm stack includes the ROCR Runtime and The ROCT Thunk Interface. ROCR runtime is built from the HSA support for AMD APUs, and extended to support discrete GPUs (dGPUs) such as FIJI series from AMD. The goal of ROCR runtime is to harness the power of GPUs for the applications by providing hardware-enabled dispatch queues for task offloading.

The ROCT Thunk Interface plays as an intermediate between the user-mode APIs and the ROCm kernel driver. The Thunk interface will iterate HSA compatible computation nodes online and refer to the resources such as global memory and LDS memory on devices.

The ROCK and ROCT together complete the runtime support of the heterogeneous computations on AMD dGPUs, as part of the ROCM software stack. The runtime components heavily depend on the existing work for HSA and supports of HSA compliant devices such as Kaveri and Carrizo APUs.

2.3.2.0.3 Heterogeneous C/C++ Compiler HCC, AKA. Heterogeneous Compute Compiler, is designed and implemented to aid the programming difficulties on heterogeneous systems. The HCC compiler, as a part of the ROCm software stack, is

based on LLVM, an active open-source compiler community. HCC implements several standards to enable programming on heterogeneous systems without touching the proprietary toolchain. By adopting the standards such as C++17, C++AMP, and OpenMP, the applications are portable access systems.

For example, HCC translates the device kernel developed with HIP onto its *parallel_for_each* routines, and therefore map to the heterogeneous computation nodes. The compiler currently supports two backends, HSA Intermediate Language(HSAIL), and the Lighting Compiler. The HSAIL backend can generate fat binaries that execute on both HSA compliant devices and AMD dGPUs; while the Lighting Compiler is specifically designed and implemented for AMD GPU ISA.

Chapter 3

Related Work

This chapter introduces the related work of programming models of multicore-embedded devices, standards library or APIs of embedded systems, and the related efforts for task-scheduling projects over heterogeneous-embedded platforms.

3.1 Parallel Programming Model

This section discusses related efforts on a higher-level parallel-programming model for embedded systems. There are some parallel-programming models available in the general computing domain. Some of the programming models are mature and widely adopted in the industry and academics. Though there is no dedicated standard per se in the embedded domain for parallel-programming; we see some efforts to implement a few programming models onto some embedded systems. This demonstrates the possibility of porting of these programming models onto embedded systems.

3.1.1 Pragma-based Programming Model

OpenMP [22] is the *de-facto* programming model for shared memory computation, and has been extended to heterogeneous architectures with the release of OpenMP 4.0. There has been some effort to mapping OpenMP onto the embedded domain. Our previous work [68] translates OpenMP to MCA Resources Management API, to support OpenMP on several Power-based embedded systems from Freescale Semiconductor. OpenMP was implemented on a high-performance DSP MPSoC, with efficient memory management and software-cache coherence in [18].

TI [58] supports accelerator features from the OpenMP 4.0 standard. Open-MDSP [33], an extension of OpenMP, was designed for multi-core DSPs to fill the gap between the OpenMP memory model and the memory hierarchy of multi-core DSPs. However, the approach was not generic enough to be used for the other systems. The authors in [70] conducted similar research. The authors discussed an OpenMP compiler for the use of distributed scratchpad memory in [44]. Some tools, which can automatically generate OpenMP directives from serial C/C++ codes, and run on a general computer and embedded system, have been discussed [73]. Marongiu et al. [45] supported OpenMP for a multi-cluster MPSoC embedded system. However, this work was not designed for portability. Thus, a fair amount of effort will be required before it can be mapped to other embedded devices. Many research activities aim to map OpenMP onto embedded systems, making OpenMP a suitable candidate for programming parallel-applications for embedded systems.

OpenACC [49] was another well-adopted pragma-based programming model used

to program to heterogeneous platforms with accelerators such as GPUs [72, 63]. However, to the best of our knowledge, there is no work on targeting OpenACC on embedded systems.

3.1.2 Language Extension and Libraries

MPI is the dominated message-passing programming model for distributed-memory environment computing [31]. However, MPI is too feature rich for embedded systems. Some projects [43, 11] implemented subsets of MPI standards to support embedded distributed systems, however, portability issues are a big hurdle for MPI to support a larger range of embedded systems.

OpenCL [32] is a language extension of C, aiming to serve as the data-parallel programming model for heterogeneous platforms. The primary use of OpenCL is for programming GPUs on general-purpose computations; though we see some work supported OpenCL on several embedded devices such as [40, 35]. OpenCL is still too low-level, and performance portability is a big issue when reusing OpenCL programs for different architectures [28].

OmpSs [25] programming model helps program on clusters of GPUs, and it has been extended to use CUDA and OpenCL recently [54].

StarPU [13] was designed for heterogeneous platforms, aiming to provide a run-time library that allows multiple parallel programs to run concurrently. However, this library mainly targets conventional computation facilities; it will not resolve the concern of portability over different embedded architectures.

There are some models, libraries, and language extensions to programming accelerators and heterogeneous systems [37] but their adaptability and suitability for embedded platforms were questionable. We have combined a high-level widely-used programming model, OpenMP, with an embedded-specific industry standard, MCA, and targeted a platform with PowerPC cores.

We believe that OpenMP has certain advantages that can be considered as an appropriate programming model for embedded systems. The OpenMP *parallel for* construct can conduct data parallelism in an efficient manner, the *task* and *taskgroup* construct allows programmers to quickly explore task parallelism for irregular algorithms. Furthermore, the newly added *target* construct brings a significant potential for support of various types accelerators for offloading computation. However, the lack of resources, different architectures, and the low-power requirements prevents the broad support of OpenMP on embedded systems. With the proposed solution framework, we can provide for embedded programmers an easy-to-adopt parallel-programming model while ensuring broad programming support for targeting embedded systems.

3.2 Standards for Programming Multicore Embedded Systems

In this section, we discuss standards and libraries of various embedded architectures.

3.2.1 Tools and Libraries for Multicore Embedded Systems

Development of software products for embedded systems typically requires implementation using the lower-level vendor-provided APIs. These do not follow standards or specifications, making the learning curve steep and software portability is not guaranteed. Adopt industry standards is crucial. It draws attention towards industry and university researchers. The Multicore Association (MCA) [4] was founded by a group of leading semiconductor companies and universities, provides a set of APIs for abstracting the low-level details of embedded software development. This eases efforts to resolve resource concurrency, communication, and task parallelism.

Architectures of embedded systems are various. A particular programming challenge is to work close to the hardware. Due to the necessity of tackling low-level details, C is used for embedded application development. Besides, vendors typically offer Software Development Kits (SDKs) that specifically fit their own devices; which makes programming embedded devices not portable and error-prone.

Language extensions have been proposed to abstract the low-level operations for a certain set of functionality on embedded systems [52, 21]. OpenCL and CUDA work closer to the platform. CUDA [48] is a popular programming model for GPU programming, but this is proprietary and limited to NVIDIA devices. Several efforts use CUDA to program NVIDIA devices such as Tegra mobile processor, for embedded applications [69, 20, 51].

3.2.2 MCA APIs

MCA APIs have been implemented and used on various embedded architectures. MCAPI was implemented on FPGA, and suitability of MCAPI for MPSoC was discussed [46]. A proof-of-concept of MRAPI on a Xilinx Virtex-5 FPGA has been applied [30]. High-level languages such as UML includes complex channel semantics and provides automatic code generation for interconnection and deployment of system components based on MCAPI [47].

Some projects use MTAPI as the task-parallelism management API for embedded platforms. Siemens recently released their MTAPI implementation as part of the *Embedded Multicore Building Blocks (EMBB)* [2] library. We adopt the *EMBB* MTAPI implementation in our OpenMP-MTAPI runtime library. The work is discussed in Section 5.

European Space Agency (ESA) built their MTAPI implementation [17] for effortless configuration of Symmetric Multiprocessing (SMP) in their space products. Stefan et al. published a baseline implementation of MTAPI for their research on open-tiled many-core SoC (System-on-Chip) [66].

3.2.3 HSA Standards

Heterogeneous System Architecture (HSA) is a consortium lead by AMD and ARM [53]; it aimed to provide hardware and software solutions to reduce disjoint memory operations for heterogeneous architectures. Unlike MCA APIs, HSA expects system-level

support for its specification, which includes both hardware and software design.

AMD Kaveri-series desktop APUs and Carrizo-series embedded/laptop APUs have been designed and manufactured in compliant of HSA specifications. The Bifrost Mali GPUs [1] also have the full support of the HSA specifications. The Mali series GPUs are widely used in smartphones and other mobile devices. Over 750 million units of Mali GPUs were shipped in 2015. There will be broader research and implementations on mobile devices for HSA.

Several academic activities were built based on the HSA specifications. A project was developed on top of the CPU-GPU hardware coherence commitment from HSA standards [50].

The programmers can use the HSA runtime library directly for application development for HSA. In addition, there were several higher-level programming tools designed to enable easier programming for HSA platforms. CLOC compiler allows users to write OpenCL-like kernels for HSA applications [3]. HIP and ROCm stack, discussed in Section 2.3, are alternative programming models for HSA compliant systems. In this dissertation, we adopt HIP as the plugin for MTAPI, to map computations onto heterogeneous-embedded accelerators.

3.3 HIP for GPU Programming

AMD announced its GPUOpen project in late 2015. Professional computing of GPUOpen provided a comprehensive open-source software stack for GPU programming. The tools provided can help with the tedious programming experiences with traditional GPU programming. By using these tools, programmers can trace problems down through the open-source stack. The details of our solution stack were introduced in Section 2.3.

3.4 Task Scheduling over Heterogeneous Systems

In general, the task-scheduling problem is known to be NP-Complete [65]. There were intensive studies, and various scheduling heuristics conducted towards efficiently scheduling tasks. Some algorithms were proposed to resolve task scheduling over heterogeneous-embedded systems.

One method used directed acyclic graphs (DAGs) to represent tasks and edges to represent the dependencies. This has been widely adopted to schedule tasks over heterogeneous architectures [55, 64, 14]. One project compares eleven static heuristics for mapping tasks over heterogeneous-distributed systems [15]. However, that work was not specifically targeted for embedded systems. As a result, some main considerations were missing such as limited hardware resources and the different system architectures.

Some work was aimed to resolve task scheduling over heterogeneous-embedded

systems. One project proposed a heterogeneous tasks-scheduling algorithm based on a DAG algorithm over networked-embedded systems [71]. This work shows a significant possibility of applying DAG heuristics to heterogeneous-embedded systems

Some papers extended the StarPU [13] runtime library for better performance. Multiple StarPU applications were composed of heterogeneous systems by introducing a hypervisor that automatically expands or shrinks contexts using feedback from the runtime system [34]. A theoretical framework and three practical mapping algorithms were proposed to achieve up to 30% faster completion time [74]. However, this work only considers the scenario of task-scheduling between CPU and GPUs. In this dissertation, we explore the performance tuning of the MTAPI scheduler with GPU plugins.

Chapter 4

GPU Plugin

GPUs have been heavily used for general purpose computing in recent years, which is mainly because of the massive-parallel architecture. GPUs can accelerate execution time from 10s to 100 times faster. With the fast development of self-driving, deep-learning, and virtual-reality industries, GPUs will play a larger role in embedded systems. However, most of the applications for GPUs are using proprietary CUDA language.

As discussed in Chapter 1, MTAPI library can be extended onto heterogeneous architectures by adding plugins. In this dissertation, we would like to extend the solution stack we proposed onto GPU devices. Thus, the choice of the plugin languages is critical. To compare the programming tools for the state-of-the-art GPU architectures, we conduct an investigation to HIP, CUDA, and OpenCL. In this chapter, we provide companions to programming models, memory models, and the performance on various GPU platforms.

4.1 Open Source Programming on GPUs

Many GPU-accelerated applications have been developed in the modern HPC and workstations. However, a big portion of those applications were programmed with the proprietary CUDA language.

Proprietary software causes many problems. End users are always tightly controlled by the vendor software. Also, end programmers do not have access to the source code of toolchain. Thus, flexibility is limited. With CUDA, the hardware options are limited to NVidia. This causes severe software portability problems for devices from different vendors, such as AMD.

Heterogeneous-compute Interface for Portability (HIP) and its tools, proposed by AMD, allow developers to convert CUDA code to common C++. Moreover, the application written in HIP can execute on both NVidia and AMD platforms. This set of tools provides GPU-programmers with more choices for hardware and development tools.

Also, HIP provides a clang-based tool called "hipify." It is to help developers quickly transform their existing CUDA-based codes onto HIP, without a lot of manual modifications. The whole software stack is completely open-sourced. Programmers can easily trace their source code through the stack and tune for better performance.

OpenCL is a cross-platform programming language for heterogeneous platforms. It is not limited to GPU, but also supports for other architectures, including CPU, DSP, and CELL. Both HIP and OpenCL can be adopted for programs on GPUs across vendors. Thus, it is desirable to compare HIP and OpenCL in this chapter.

Overall, HIP has certain advantages for programming on GPUs. Firstly, with HIP, the programmers can write both host and devices code in C++. Moreover, the compiler supports the latest C++17 features such as templates, lambdas, or classes. OpenCL requires compilation of the kernel at the run time, which is not a convenient way, and can potentially cause many problems. The HIP compiler compiles both the host code and the device-kernel code offline, creating a single fat binary for execution.

Secondly, the grammar of HIP is less complicated. Programmers can move from their familiar programming languages onto HIP without a steep learning curve. Also, HIP is developed as a part of the open source software stack by AMD. The programmers have significantly more flexibility to track throughout the whole software toolchain. It is very important to provide industry standards and open source toolchains to programmers.

4.2 Performance Evaluation of HIP

We evaluate the performance of HIP over general computing GPU platforms from both AMD and NVidia. The motivation behind is to make sure HIP can achieve good performance. This is the prerequisite for HIP to be adopted for heterogeneous computing. In addition, it is worthwhile to investigate HIP on GPUs across vendors.

We have configured test platforms from both AMD and NVidia. The test bed from AMD features a Fiji Nano GPU card and a test bed from Nvidia equipped with one GTX Titan X GPU. Those GPUs are all high-end consumer-grade, which is powerful for general computing. The main features for AMD Fiji NANO is as

follows:

- Featured 4GB HBM (high-bandwidth memory).
- Global-memory bandwidth is 512 GB/sec.
- 4096 streaming processors.
- 8192 GFLOPS computation capacity.
- TDP: 175W.

And here is features for NVidia Titan X GPUs:

- Featured 12 GB DDR5 global memory.
- Global-memory bandwidth is 336 GB/sec.
- 3072 CUDA cores.
- 6144 GFLOPS computation capacity.
- TDP: 250W.

The hardware capacity for these two GPUs is on par. The major difference is the configuration of the global memory.

FIJI Nano GPU is one of the first GPUs that features the HBM memory. It delivers very high memory bandwidth, a rate at 512 GB/second. However, due to the limitation of the new HBM manufacture technology, there are only 4 GB HBM

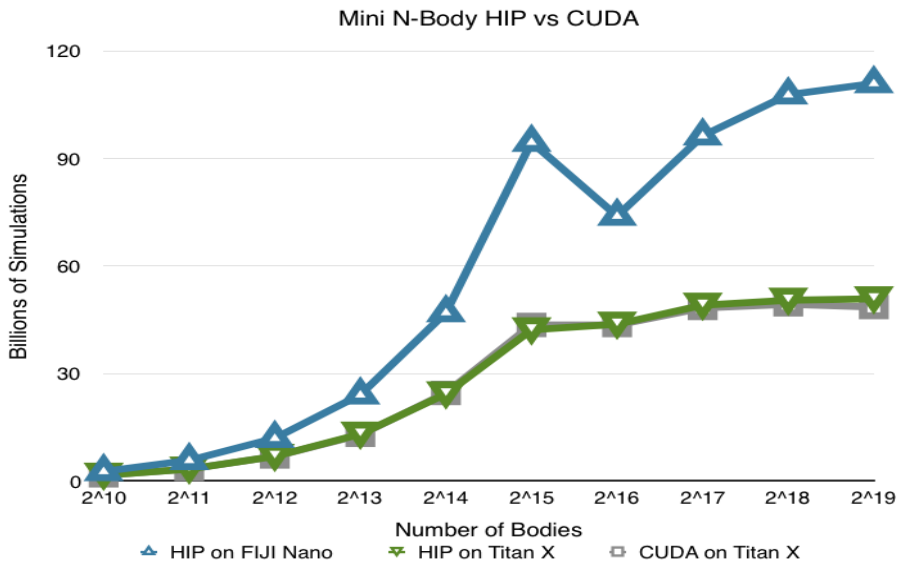

Figure 4.1: Mini N-Body simulation performance comparison

on the device. It is quite tight for general purpose computing. The Titan X GPU features much larger 12 GB DDR5 memory. However, the memory bandwidth is much smaller compared to the FIJI Nano GPU. We believe these two GPU cards are good candidates for our performance evaluation.

The benchmark simulates a n-body algorithm and has been ported to HIP. We executed it on both AMD FIJI NANO GPU and NVidia Titan X GPU. Also, with HIP implementation, the application can perform as good as that implemented in CUDA.

As seen in Figure 4.1, FIJI Nano achieves better performance, with the help of higher computation capacity, and wider global-memory bandwidth. The overall performance on the FIJI Nano platform is much greater than that on the Titan

X GPU card. The difference in performance is greater along with larger size of input. Moreover, HIP and CUDA achieves similar performance levels on the Titan X platform. On the NVidia platforms, HIP only plays as a very thin layer that translates the HIP routines back to CUDA routines.

HIP has the potential for being the programming model for many GPU devices, and it is fully open-sourced. Thus, we choose to develop HIP as the plugin target for our heterogeneous MTAPI library. Moreover, we extend the use of HIP onto heterogeneous-embedded systems with our solution stack.

Chapter 5

MCA API Work

In this chapter, we introduce the work we have done towards extending and enriching MCAPI and MRAPI libraries. We study and implement MCAPI onto a heterogeneous-embedded board, and extend MRAPI with more thread-level resource-management capacity. We secondly introduce the work that extends the MTAPI library onto heterogeneous-embedded systems, which is achieved by defining external plugins of HIP. Applications written in our MTAPI library can execute on embedded systems with GPUs across vendors.

5.1 Implement MCAPI on Heterogeneous Platform

Our goal is to design and create a portable programming paradigm for heterogeneous-embedded systems. We plan to leverage the recently ratified accelerator features of the de-facto shared-memory programming model OpenMP that may serve as a vehicle for productive programming of heterogeneous-embedded systems. To begin with, we have studied the industry standards MCA API, which specifies essential application-level semantics for communication, synchronization, resource management, and task management capabilities.

We have explored the multicore-communication APIs (MCAPI) that are designed to capture basic communication and synchronization required for closely distributed embedded systems. We have considered Freescale QorlQ P4080 as the target and evaluation platform for this work. We identified the primary challenge of this project is to establish a communication mechanism between the host processor and the Data-Path-Acceleration Architecture (DPAA) incorporating accelerators, which includes *Security Engine* (SEC 4.0), and *Pattern Matching Engine* (PME) accelerators. The work for this section has been published in [59, 60].

5.1.1 Target Platform

The P4080 development system is a high-performance development platform supporting the P4080 Power-architecture processor. Figure 5.1 shows the preliminary

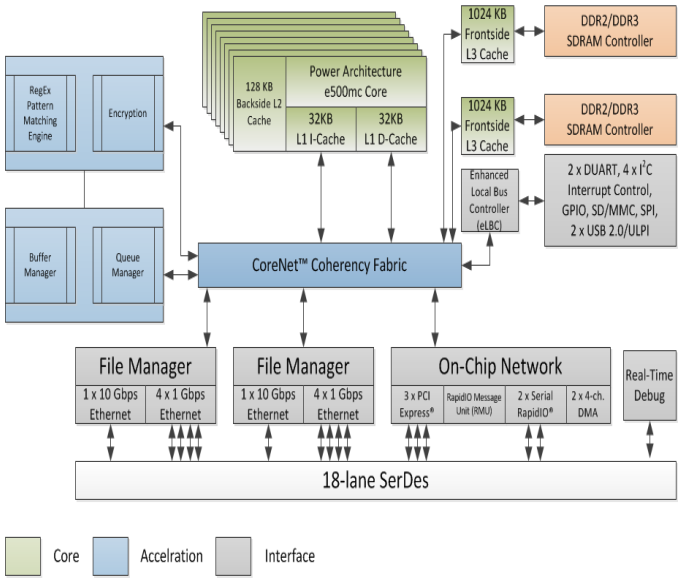

Figure 5.1: P4080 block diagram

block diagram of P4080.

5.1.1.1 P4080 Processor

The P4080 processor is based upon the *e500mc* core built on Power Architecture and offering speeds at 1200-1500 MHz. It consists of a three-level cache hierarchy with 32 KB of instruction and data cache per core, 128 KB of unified backside L2 cache per core, as well as a 2 MB of shared front side cache. There are totally eight e500mc cores built in the P4080 processor. It also includes the accelerator blocks known as Data-Path-Accelerator Architecture. It offloads various tasks from the host, including routine packet handling, security-algorithm calculation, and pattern matching. The

P4080 processor has been used for combined control, data path, and application-layer processing. It is ideal for applications such as enterprise and service provider routers, switches, based station controllers, radio network controllers, and long-term evolution (LTE). It also has been deployed in general-purpose embedded computing systems in the networking, telecom/datacom, wireless infrastructure, military, and aerospace.

5.1.1.2 DPAA

The P4080 includes the first implementation of the PowerQUICC *Data-Path-Acceleration Architecture* (DPAA). This architecture provides the infrastructure to support simplified sharing of networking interfaces and accelerators by multiple CPU cores. DPAA includes the following major components:

- Frame manager
- Queue manager
- Buffer manager
- Security engine
- Pattern matching engine

DPAA plays an important role in addressing critical performance problems in high-speed networking I/O. It provides a bundle of user space APIs to be called by the end user to customize accelerator parameters and configurations.

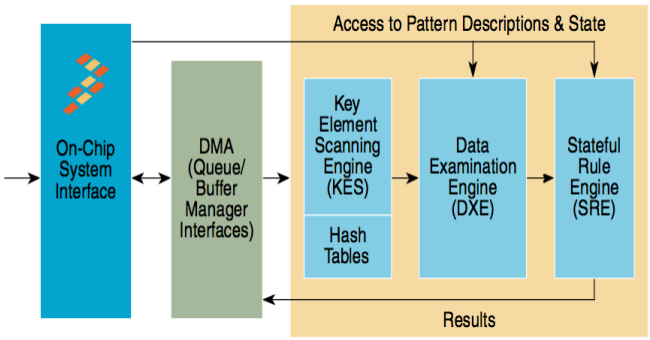

Figure 5.2: P4080 PME

5.1.1.3 Pattern Matching Engine (PME)

The PME provides hardware acceleration for regular expression scanning, scan across streamed data and finds out the matched patterns, providing a very high-speed hardware acceleration. Patterns that can be recognized or matched by the PME are two general forms, *byte patterns*, and *event patterns*. Byte patterns are single matches such as 'abcd123' existing in both the data being scanned and in the pattern specification database. Event patterns are a sequence of multiple byte patterns. In the PME, event patterns are defined by stateful rules. The PME specifies patterns as regular expressions (Regex). The host processor needs to convert Regex patterns into the PME's specification data path, which means there is a one-to-one mapping between a regular expression and a PME byte pattern. Within the PME, match detection precedes in stages. The *Key Element Scanner* performs initial byte-pattern matching, with the handoff to the *Data Examination* engine for an elimination of

false positives through more complex comparisons. The *Stateful Rule Engine* receives confirmed necessary matches from the earlier stages, and monitors a stream for addition subsequent matches that define an event pattern. Figure 5.2 shows the general block diagram of PME.

We explored and analyzed DPAA and its component PME in depth. We intend to simplify the programming-interfaces to the low-level details of PME by exploiting the capabilities of MCAPI. This API has the vocabulary to establish communication on a bare-metal implementation (no OS at all), that was one of the motivational aspects to choose PME as our candidate accelerator. This hardware accelerator does not offer shared memory support with the host processors and the power processor cores (e500mc cores) for the P4080.

As a result, the data movement between the host and the accelerators need to be handled via a *DMA channel* explicitly. Thus, it is desired to use a standard communication API such as MCAPI to handle the data transportation and messaging.

5.1.2 Design and Implementation of PME support in MCAPI

Firstly we studied functionalities and the prototype implementation of MCAPI. MCAPI is an industry-standard API for inter-core communication within a loosely coupled distributed embedded SOC. It can be treated as MPI. However, the APIs of MCAPI are much lighter than those of MPI. And MCAPI is designed for the embedded platforms. Nodes are a fundamental concept in the MCA interfaces that map to an independent thread of control such as a process, thread, or processor.

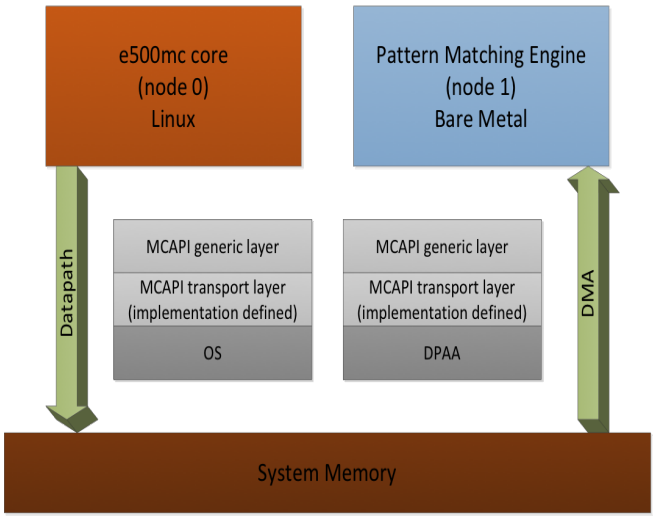

Figure 5.3: Solution stack

In our implementation phase, we set up the PME as a node in MCA and e500mc processors as the other node. Our solution stack is seen in Figure 5.3.

We explored the MCAPI transportation layer that serves as the communication layer for PME. We modified the 'create end point' function to cater to the PME platform. We also modified two transportation layer APIs to build the MCAPI transportation channel for PME. They appeared on top of the DMA channel for PME package transportations.

For the MCAPI message passing, we used the Pattern Matcher Control Interface (PMCI) library, a C interface to send and receive pattern matcher control commands to the PME. We found that this functionality is partly abstracted by utilizing the

MCAPI massage mechanism, including sending and receiving messages. To implement PMCI support within MCAPI transportation layer, we added the PMCI shared library into the MCAPI build system. In this connectionless-messaging mechanism, the source code makes function calls directly to the PMCI library. The primary functions involved in this process include:

- *pmci_open*
- *pmci_close*
- *pmci_set option*
- *pmci_read*
- *pmci_write*

For instance, the routines to open *pmci* was mapped to the MCAPI transportation-layer-initialization subroutine. The routine to close *pmci* was included in the MCAPI transportation-finalize subroutine. The read and write routines have been mapped into the MCAPI message receive and send subroutines while the routine to set configuration has been incorporated into the MCAPI node and endpoint initializations.

As seen in Figure 5.4, we set the Power-processor as one node with a dedicated endpoint, and the PME as the other node with another endpoint. We therefore encrypt PMCI message under the MCAPI communication interface. Unlike the package channel or the scalar channel in MCAPI, sending and receiving messages in MCAPI are connectionless, which means there is no need to build connections between the nodes beforehand. This enables the flexibility to send or receive messages

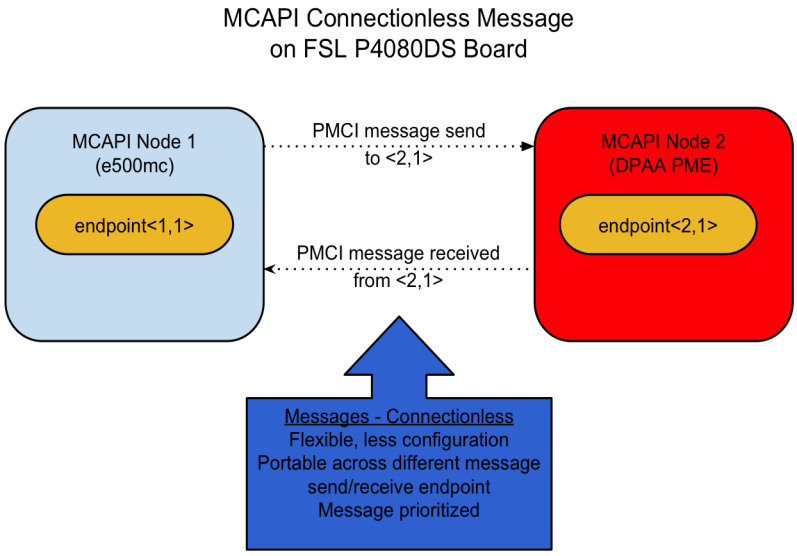

Figure 5.4: MCAPI on PME

between nodes and endpoints, as well as to communicate with multiple endpoints, simultaneously. Another important feature provided by MCAPI is the capacity to manage prioritized messages, which has potential for being utilized in our future work.

5.2 Multicore Resource Management API extension

MRAPI supports process-level parallelism by mapping MRAPI nodes on processes and utilizing the MRAPI synchronization primitives to synchronize between nodes. However, such parallelism can be cumbersome for parallelizing embedded systems. The overhead due to launching a process and inter-process communication (IPC) (causing additional context-switching) can be a performance kill. Each process has a private address space; thus one process is unable to access the other process's data. Unlike processes, threads are lightweight. The threads lower the cost of creation and the ability to exchange large data structures by passing pointers. OpenMP writes multithread applications without creating, synchronizing, or destroying threads. MRAPI can enhance the thread-level support for embedded platforms.

5.2.1 Target Platforms

For this work, we have chosen the T4240RDB platform from *Freescale*'s QorlQ family. Since this hardware platform is not a typical X86 platform, we believe it is important

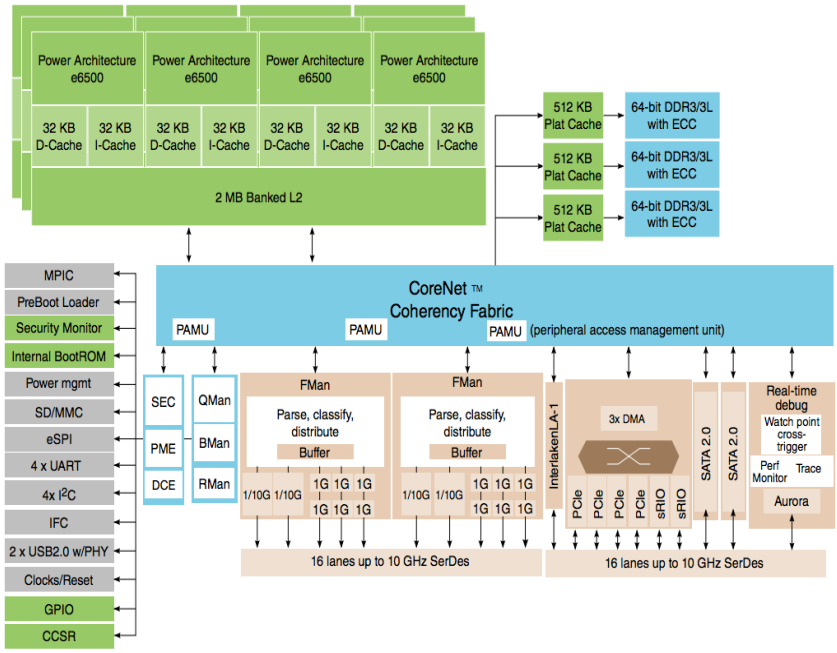

Figure 5.5: T4240RDB block diagram

for the reader to know about the features of the platform and its system setup procedures, before we discuss the design and implementation of the software.

5.2.1.1 T4 Processor

The T4240RDB platform features twenty-four virtual threads from twelve PowerPC e6500 cores, running at 1.8 GHz and providing rich I/O capabilities. The Freescale T4 processor family is commonly used in networking routers, switches, gateways to fully utilize the combined control, datapath support and application layer processing, and also for general-purpose embedded computing systems. Twelve PowerPC e6500 64-bit dual-threaded cores with integrated AlitiVec SIMD processing units are clustered

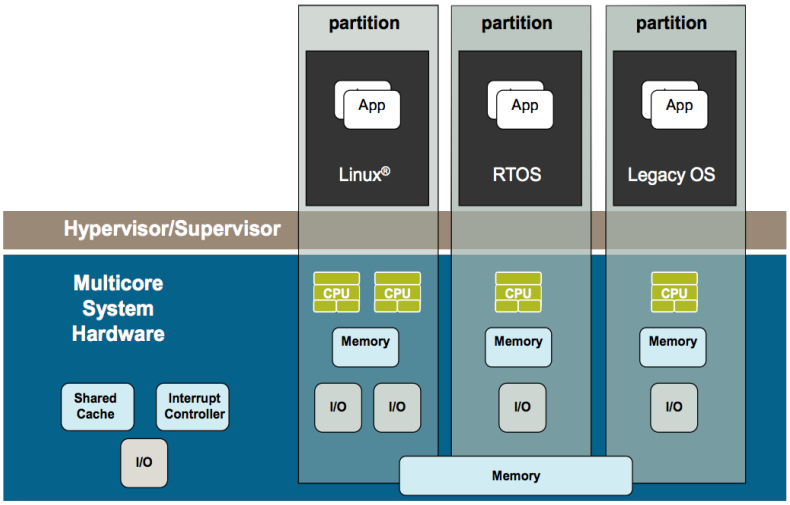

Figure 5.6: T4240RDB hypervisor

in the T4240RDB board, manufactured using 28 nm process. The e6500 core includes a 16 GFLOPS AlitiVec technology execution unit that supports SIMD architecture, to achieve DSP-like performance for math-intensive applications and is considered to be mapped to the OpenMP 4.0 SIMD support. E6500 cores are clustered to four cores with a shared multibank L2 cache, and three groups in the T4240RDB board are connected by the CoreNet coherency fabric, sharing a 1.5 MB CoreNet (L3) cache. Additionally, it has hardware support for L1 and L2 cache coherency. The design of e6500 cores also uses several low-power techniques, including pervasive virtualization and cascading-power management.

T4240RDB provides support for hypervisor for embedded-systems. The Freescale embedded hypervisor can add a layer of software that enables efficient and secure partitioning of a multicore system, including the partition of a system's CPUs, memory

and I/O devices. Each partition capable of executing a different or the same guest operating systems. We plan to use MCAPI to exploit hypervisor in future work. Figure 5.6 illustrates how Freescale Hypervisor manages the embedded-systems hardware and partitions on the system. The 12 e6500 cores in T4240RDB uses the Freescale Hypervisor, so we can efficiently simulate an Asymmetric Multiprocessing environment in many formats.

We consider T4240RDB as a heterogeneous platform. We partition the T4240RDB board in two parts that each has own hardware resource, including CPUs, memory and I/O devices. As seen in Figure 5.6, we can explore our OpenMP runtime based on MCA libraries when the partitions use different operation systems.

5.2.1.2 T4240RDB Setup

Unlike the general-purpose computer, the T4240 RDB uses a time-consuming procedure to boot an embedded device and configure it for a specific term. We describe the procedures for setting up the T4240RDB board to illustrate how to set up an embedded platform.

The T4 board comes with a pre-installed u-boot and the embedded Linux image on the NOR flash-drive. Moreover, by default, the T4 boots from the NOR flash drive. In the default configuration, the file system is refreshed for every reset. However, the default configuration required modification.

The system initially loads the u-boot bootloader, and then reads the Linux image onto RAM, therefore, the Linux kernel in RAM. Any changes we made to the board,

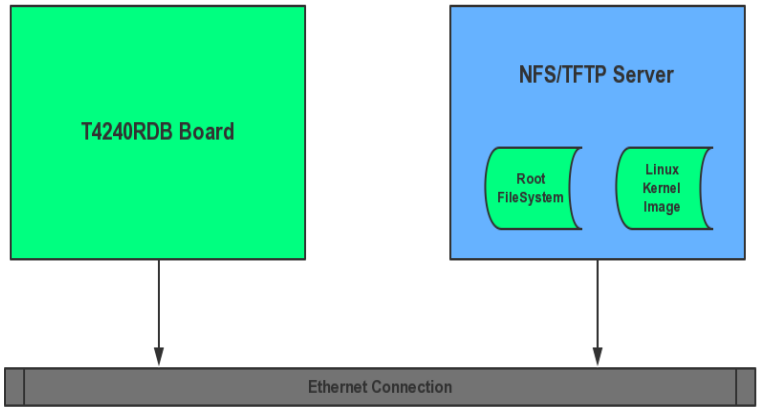

Figure 5.7: NFS development environment

would be lost after the system reset. We can not connect the board to Ethernet, which makes file transfer difficult and development impossible.

We configured the T4240RDB to load the embedded kernel image from a TFTP server while u-boot mounted an NFS root file system, as seen in Figure 5.7. Trivial File Transfer Protocol (*TFTP*) is a file transfer protocol that allows a client to access a file at a remote host [57]. The TFTP server is configured in one Linux desktop while the T4240RDB board is the client that obtains the image files in u-boot. NFS [56] is a distributed file system protocol that allows a user to access data over a network much like accessing local storage. We also configured the Linux desktop to host the NFS server. The root file system is customized and saved on the remote NFS server, while not using the limited local-hardware resources on the board.

5.2.1.3 Comparing T4240RDB with P4080DS

We highlight the differences in the target platform that we had used for our in Section 5.1 and the platform that we are using for the work discussed in this subsection. Our goal is to provide a software toolchain that can be used across more than one platform, and we have substantially improved our previous toolchain to make it suitable platform. The P4080DS processor with eight Freescale e500mc PowerPC cores, is compliant with PowerISA v.2.06 and includes hypervisor and visualization functionality. It supports CoreNet communications which connect cores and data-path accelerators. Eight e500mc cores are attached to the CoreNet fabric directly, unlike the e6500.

T4240RDB platform with twelve PowerPC cores connect four e6500 cores as to the CoreNet fabric. The L1 cache is 32 KB, the same for both processors. While the L2 cache size is 128 KB.

5.2.2 MRAPI Node Management Extension

MRAPI's node initialization process was used to create threads associated with MRAPI node IDs. MRAPI node initialization process created new nodes related to node IDs and registered the node related information in the global MRAPI database shared by all the nodes in one domain. MRAPI categories the hardware resources into four categories: computation entities, memory primitives, synchronization primitives, and system metadata. This rich feature set of MRAPI allows process-level and system-level resource management. MRAPI supports a team of nodes where one node

is the host and the other are accelerators. Our work procedures simplify MRAPI on multi-thread applications, as well as the OpenMP runtime libraries. The MRAPI reference implementation was modified to accommodate the extension of this project as seen in Listing 5.1.

```
typedef struct{
    pthread_t *thread_handle;
    pthread_attr_t *attr;
    void *(*start_routine) (void *);
    void* arg;
} mrapi_thread_parameters_t;

void mrapi_thread_create(
   MRAPI_IN int domain_id,
   MRAPI_IN int node_id,
   MRAPI_OUT mrapi_thread_parameters_t* init_parameters,
   MRAPI_OUT mrapi_status_t* status )
 {
     if (mrapi_impl_initialized()){
         if(mrapi_impl_thread_create(domain_id, node_id, init_parameters))
             *status = MRAPI_SUCCESS;
     }
    else{
       *status = MRAPI_ERR_NODE_NOTINIT;
    }
 }
```

Listing 5.1: MRAPI Node Extension

The thread-creation operation is accomplished for each node calling *mrapi_thread_create*. The function created a worker thread for the node requested and registered thread-related information in the global-domain database for the calling node. This is associated with the created thread and managed for later use.

5.2.3 MRAPI Memory Management Extension

MRAPI shared memory constructs maps the memory allocation onto the system-level shared memory, which is a function of the Inter-Process Communication (IPC). However, this is not suitable for OpenMP and the other thread-level parallel computation. We extended the MRAPI implementation to offer end-users more memory allocation choice. For instance, most of the OpenMP global-shared data is mapped to the process private heap instead of the system-level shared memory. This allows sharing among threads. Data mapped onto the heap can be shared by all threads created by the same process.This facilitates the global-data movement. We make it more feasible to utilize varying memory features available on different platforms, by extending the MRAPI memory model to support thread-level memory.

5.3 Extend MTAPI onto heterogeneous-embedded Systems

In Chapter 5, we present work that uses low-level industry standard, MCA APIs for multicore-embedded systems. We further discuss the abstraction of the software stack to use a higher-level pragma-based approach such as OpenMP that is further translated to the low-level standard MCA API.

However, as the concept of Dark Silicon [27] draws a big attention by the hardware system designer to rethink the scalability of multicore system, the trend towards heterogeneity in embedded systems is very evident.

Dark silicon is a term used in the electronics industry. In the nano-era, transistor scaling and voltage scaling are no longer in line with each other, resulting in the failure of Dennard scaling [29].

Besides good performance has been achieved by adopting heterogeneous computation, the power efficiency is another big advantage for heterogeneous computation. A significant portion of embedded systems is restricted in power consumption, especially for the Socs that need to be powered by a battery.

Thus, in this chapter, we discuss the work to extend both standard APIs and the OpenMP programming model in the heterogeneous-embedded systems, specifically targeting embedded systems with GPU.

5.3.1 Design and Implementation of Heterogeneous MTAPI

In Section 2.2.3, we discussed the MCA Task Management APIs (MTAPI) and its key features. In Subsection 6.2 we explore task-parallelism using OpenMP and MTAPI over broad coverage of embedded systems. We further explore Heterogeneous System Architectures (HSA) as a target platform for the MTAPI library, by adopting HIP as the plugin for the MTAPI RTL system. This work enables the support of heterogeneous computing for MTAPI library.

5.3.1.1 Heterogeneous System Architectures (HSA) Background

A short background of HSA and its features are discussed. The HSA Foundation is a not-for-profit organization of industry and academia, to define and develop Heterogeneous System Architecture (HSA) which is a set of open-sourced computer hardware specifications and software-development tools. The founders of HSA include AMD, ARM Holdings, Imagination Technologies, MediaTek, Qualcomm, Samsung, and Texas Instruments. They are committed to producing HSA compatible products to enrich the HSA ecosystem. As a long term goal, HSA will cover the productions from heterogeneous-embedded systems, portable devices, personal computers, and up to high-performance computers. Software that uses HSA standards can be executed on different architectures without refactoring.

5.3.1.2 Development Platform

The development machine is powered by an AMD A10-7850K-2, a Kaveri APU. The APU contains four CPU cores and 8 AMD R9 GPU cores. The L1 cache is 256 KB while there is total 4 MB L2 cache. The Kaveri APU is the first product that features HSA technology. HSA is new technology and related support of developing HSA is limited. The vast potential of HSA encourages us to implement our system on this cutting-edge heterogeneous platform.

5.3.1.3 heterogeneous Uniform Memory Access

The HSA proposed heterogeneous Uniform Memory Access (hUMA), which allows the CPU to pass a pointer to a GPU directly without explicit data movement. After the GPU finishes execution, the CPU can immediately read the buffer. Data copy between the CPU and GPU are no longer required. The key features of hUMA is as follows:

- *hUMA* supports Bi-directional Coherent Memory. CPU and GPU must have the same view of the entire memory space without explicit data movement to keep memory coherence.
- The GPU can access pageable memory from virtual memory that is not in physical memory.
- Both CPU and GPU can dynamically access and allocate any location in the system's virtual memory space.

Applications for GPUs suffer by slow data-movement. hUMA plays a significant role to HSA, and it has the potential to be a standard for future heterogeneous computation.

5.3.1.4 HSA Context Switch Support

The context switch was not supported in GPU architecture, due to the lack of hardware support. HSA proposed a software alternative to this problem. The HSA com

devices provide a decent way for programmers to explore irregular applications on GPUs. This concept will play a bigger role in the evolution of the architecture for future GPUs.

5.3.2 Motivations and Design to Support HIP as Plug-in to MTAPI

MTAPI is a standardized API for expressing task-level parallel programming on a broad range of embedded systems with different hardware architectures. The MTAPI specification is designed for both homogeneous and heterogeneous systems. HSA aims to provide easy-to-use programming interfaces for heterogeneous systems, with the support of different kind of devices, including CPU, GPUs, DSP, or FPGAs. Also, it provides a simplified memory model that connects these devices.

In contrast to HSA, MCA APIs target to serve as the comprehensive industry standards that fully simplify and abstract the underneath system resources, enabling the power of portability over a set of embedded systems. Thus, it is interesting to investigate the potential of combining the portability and abstraction enabled by MCA APIs and the heterogeneous coverage provided by HSA and HIP.

Also, the HSA Foundation targets to encourage and promote HSA-provided devices to cover the systems from embedded, portable devices to workstations and supercomputers, to dominate the standard for heterogeneous computation. Moreover, most of these company-members of the HSA Foundation focus on Embedded Systems. It is very possible that in the near future the HSA standard will be widely

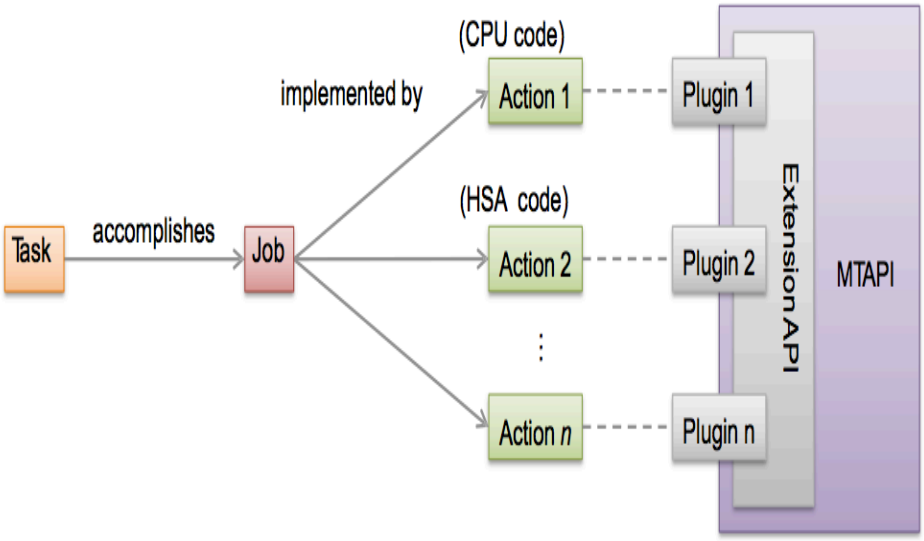

Figure 5.8: Plug-ins for MTAPI actions

supported in heterogeneous multicore-embedded systems. Starting with the release of ARM Mali GPU naming in Bifrost, the HSA feature has been supported. Due to the large number of the Mali devices (used 700 million in 2015), HSA will play a much broader role in GPU computing. All the trends and evolution encourage us to further extend our OpenMP-MCA work, to make HIP and HSA a plug-in for heterogeneous-computing.

MTAPI specification is designed to take heterogeneity into consideration, in particular with the original definition of Jobs, Tasks, and Actions. In MTAPI, each MTAPI task is bundled with a job. The job is a piece of the processing implemented by an action. Also, one job can be implemented by multiple actions; those actions can be either software-defined or hardware-defined. As seen in Figure 5.8, MTAPI actions can take plug-ins from the external environment.

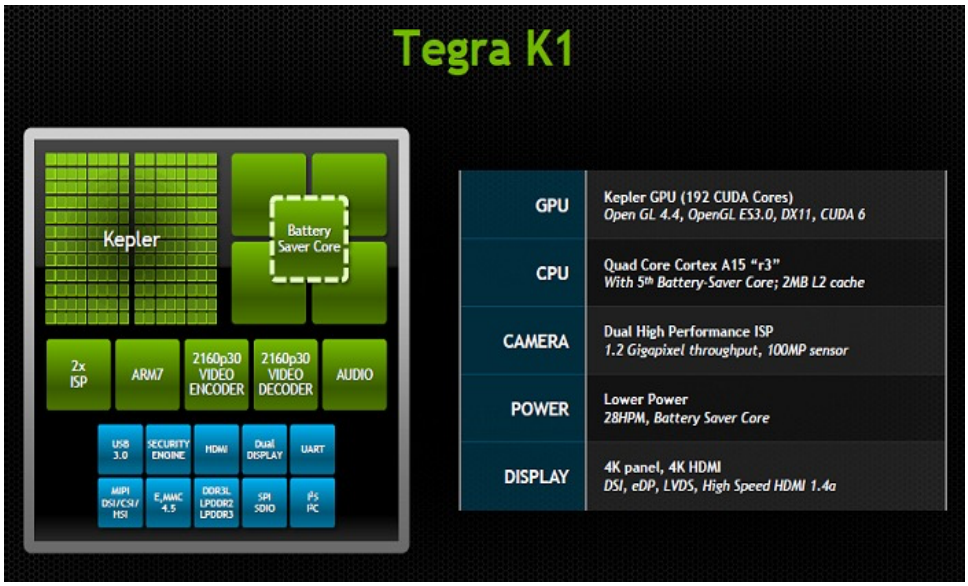

Figure 5.9: NVidia Tegra K1 features

In this dissertation, we have extended the current MTAPI reference implementation [2] to take actions that are implemented by HSA, targeting NVidia GPUs, AMD HSA-compatible APUs, and AMD dGPUs.

5.3.3 Performance Evaluation Platforms

To measure the performance of our heterogeneous MTAPI implementation, we set up two state-of-the-art embedded systems that feature GPUs for exploration and developed a set of benchmarks.

5.3.3.1 NVidia Jetson TK1 Board

As seen in the Figure 5.9, the Jetson TK1 embedded development-board features a NVidia Tegra K1 processor, which includes a ARM quad-core Cortex A15 CPU and 192-core Kepler GPUs. It is manufactured in 28 nm technology. The board also includes 2 GB of DDR3L DRAM (2 ways) and 16 GB of fast eMMC for storage. On the board, both of the CPU and GPU cores use the same physical DRAM memory. The theoretical memory bandwidth of TK1 board is 14.928 GB/sec.

However, unlike an HSA-compliant device, its CPU and GPU has non-unified memory spaces. Also, there is not full hardware-cache coherency between the CPU and the GPU. As a result, the applications still need memory-copy between the host CPU and the device GPU. We have several performance evaluations that illustrate the performance impact of those architecture features. We observed longer times spent on data-movement on the TK1 board than on the Carrizo board.

The TK1 board has been widely adopted for both research and industry, including the Computer-Vision related domains, auto-piloted cars, and robots. Overall, the TK1 board is a very powerful embedded system. The CUDA Toolkit from NVidia is supported by the board. Thus, it is a good candidate for us to test the heterogeneous MTAPI.

5.3.3.2 AMD Carrizo Embedded Board

As the very first device that is fully compliant with HSA 1.0 specifications, Carrizo was released in mid-2015. The Carrizo APU used was manufactured in 28 nm.

It contains 4 "Excavator" CPU cores and 512 GCN streaming processors for an integrated GPU. The Carrizo board is configured to run with two channels of DDR3-1600 memory, and each channel of the memory has 12.8 GB/sec theoretical bandwidth. Thus, the theoretical-memory bandwidth of the Carrizo processor is 25.6 GB/sec.

The Carrizo APU processor and its SoCs have been widely used in the embedded-system's domain, including medical-imaging equipment, industrial controllers, and gaming. Moreover, most of the embedded applications take full advantage of the powerful GPU cores by using OpenCL toolchain. With the release of HIP and ROCm, it is desirable to program with such tools, for better performance and easier programming interfaces. We set up the heterogeneous environment MTAPI and the HIP on the board, and achieved a very competitive performance.

5.3.4 Performance Evaluation of Extended MTAPI Implementations

We evaluated the performance of the MTAPI implementation by using one set of GPU-specified benchmarks and one set of real applications. The first set of benchmarks was purely designed for platforms that are equipped with GPU devices, to test the features that are of most interest in the GPU domain GPU-streaming rate, global-memory bandwidth, and PCIE data-transfer rate. All benchmarks were ported to our platforms, with HIP as the plugin.

We also ported a set of real applications and benchmarks to compare our solution

stack for embedded systems with GPUs. The applications and benchmarks had OpenMP running on the host CPUs and HIP on the GPUs. We compared the performance of the host side and the GPU-accelerator side, across the platforms from AMD and NVidia.

5.3.4.1 GPU Benchmarks Comparison

We focused on comparing the fully optimized powerful computation rates that target GPU devices in the embedded systems. Therefore, the performance was for the two heterogeneous-embedded systems we set up previously: the NVidia TK1 development board and the Carrizo board.

5.3.4.1.1 GPU Streams Benchmark The memory bandwidth of GPU devices is significantly larger than the traditional CPU cores. We saw a very wide, 512 GB/sec on the AMD Fury series discrete-GPUs. These were equipped with the High-Bandwidth Memory (HBM), compared to 51.2 GB/sec for an E5-2687 Workstation CPU processor. We tested the efficiency of benchmarks for streaming-applications. The GPU-STREAM [24] benchmark was ported and we measured the performance on our test beds. The benchmark essentially tests the performance of four simple-kernels on GPUs:

- *Copy: c[i] + a[i]*
- *Multiply: b[i] = n*a[i](n stands for a constant)*
- *Add: c[i] = b[i] + b[i]*

- *Triad: a[i] = b[i] + n*c[i]*

Those kernels require minimum time on float-variable computation, leaving the performance-bottlenecks to be the global-memory bandwidths of the underlying GPU system. The performances seen in Table 5.1 were measured in GB per second. We showed the theoretical memory-bandwidth of Carrizo was 25.6 GB/sec, and that of TK1 board was 14.928 GB/sec.

Table 5.1: GPU-STREAM performance evaluation

Kernels	**Copy**	**Multiply**	**Add**	**Triad**	**Efficiency**
Double on Carrizo	21.552	21.630	21.422	21.359	84.00%
Float on Carrizo	18.825	18.769	21.389	21.426	78.31%
Double on TK1	12.908	12.933	13.111	13.097	87.16%
Float on TK1	10.727	10.672	11.578	11.732	74.8%

As illustrated in Table 5.1, the TK1 board achieved better efficiency compared to the Carrizo board on double-precision and slightly worse on single-precision. However, the Carrizo board with a much larger theoretical memory-bandwidth makes the overall performance better than the TK1 board.

5.3.4.1.2 GPU Stride Memory Access Benchmark It is important to use the coalesced memory-access pattern to achieve optimal performance on GPUs. However, the coalesced memory-access pattern is not guaranteed. We normally expect stride memory-access pattern for GPU applications, such as accessing an element in data-structures. Thus, it is important to measure the performance of the GPU devices on the stride memory-access pattern.

We adopted the benchmark provided from [9], and ported it to run on both TK1

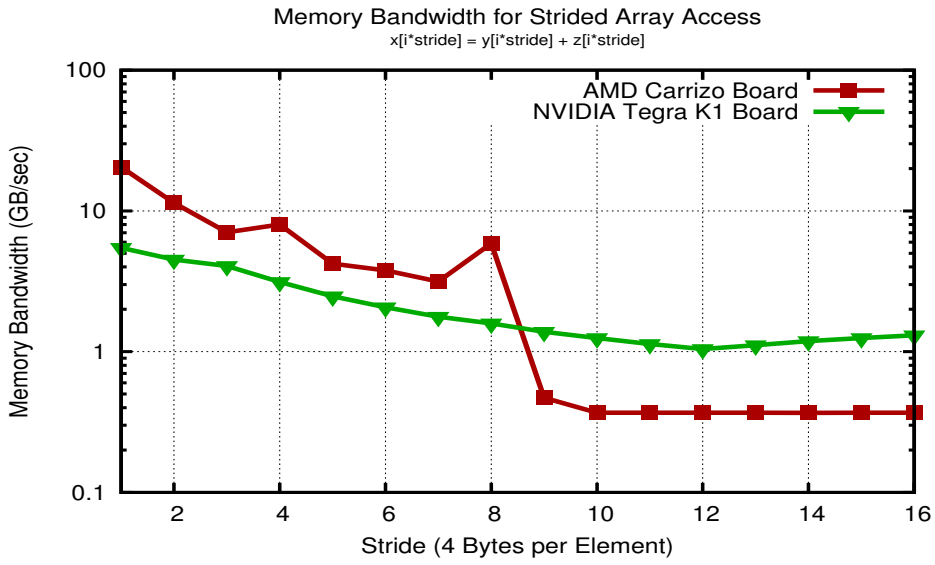

Figure 5.10: Stride memory access benchmark results

and Carrizo embedded-boards.

As seen in Figure 5.10, the performance on the Carrizo board had a certain advantage when the size of the stride was less than 8 bytes. However, when the size of the stride increased to more than 8 bytes, the performance on Carrizo decreased dramatically. The behavior relates to the wavefront-size of the AMD GCN architecture and the optimizations on the driver-level support.

For GPU architecture, *warp* for NVidia GPU and *wavefront* for AMD GPU are the basic units of hardware-scheduling. The warp for NVidia GPUs consists 32 threads and the wavefront for AMD GCN GPUs has 64 threads. Each of the warp/wavefront for GPU applications processes a single-instruction over all of the threads at the same time. Thus, when the stride size is larger than 64 bits (8 bytes),

there is no data reuse inside the wavefront for each read from global-memory. It can explain the poor performance when the stride size was larger than 8 bytes for the Carrizo board.

In comparison, the performance on the TK1 board was more stable on a different stride pattern. Good performance should be related to the optimizations on the NVidia toolchains and drivers for the stride-memory and texture-memory access. Because the CUDA toolchain is proprietary, we had no way understand it at a deeper level.

The takeaway from this benchmark evaluation is that, for the Carrizo board, the size of stride memory-access should be limited to less than 8 bytes or 64 bits to achieve good performance. Moreover, AMD drivers have room to improve, for large-stride memory-access.

5.3.4.1.3 Mixed Benchmark for GPUs In the previous GPU-specific benchmarks, we evaluated the data-streaming and the stride-memory access rates, to show several characteristics of the two systems. Additionally, we ported the mixbench [38] to measure the performance of GPUs on a mixed operational-intensive kernel. The benchmark contains mixed-operations of multiplication and addition. It was configured for the data types including float, double, and integer.

As the iterations increased for each set of tests, more data was fed into the computation-kernels. As seen in Figure 5.11, the Carrizo board has a significant increase in the performance for each of the float, double, and integer data types. For float-type benchmark in Figure 5.11a, a stable performance for iterations smaller

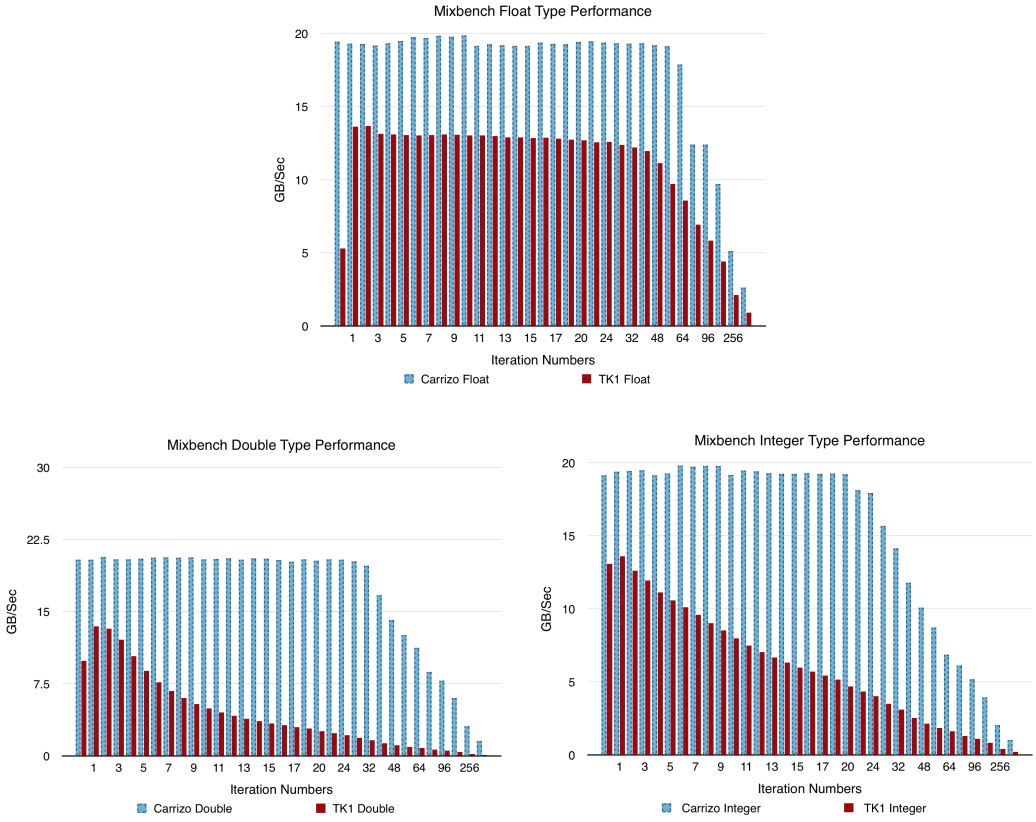

Figure 5.11: Evaluating mixbench benchmark for Carrizo and TK1 Boards

than 32 was seen. Both on Carrizo and TK1, the performance started to decrease at similar rates for iterations larger than 64. On double-precision seen in Figure 5.11b and integer type seen in Figure 5.11c, the performance on the Carrizo board is very stable on iterations smaller than 32. However, it starts to decrease gradually for increasing iterations.

For the TK1 board, the performance decreased as early as the second iteration. The decreased rate for double-precision was similar to the normal-distribution, and the rate for integer-type was similar to a linear decrease. The performance advantage on the Carrizo board is the efficient HSA-compliant architecture. Those features include *hUMA* memory for faster data-coherency and copy rates, as well as the hardware kernel-dispatch. Those features helped minimize the overhead for kernel-executing on the GPU cores, resulting in better performance.

By observing the three GPU-specific benchmarks ported to our solution stack, we concluded that TK1 board from NVidia is very powerful. The performance for this platform is more stable with different input sizes, because of to the highly optimized CUDA toolchains. However, proprietary-software prevented us from investigating the performance we have observed from the benchmarks.

With the HSA-architecture, we expected good performance on the Carrizo board. However, HIP is relatively new and not as mature as NVidia CUDA toolchains. We see more room for improvement for the compiler and driver. We have to emphasize that the HIP and ROCm stack are all open-sourced. Unlike CUDA toolchains, we can explore the stack and tune performance as needed. Industry standards and open source are the keys for general-computing on heterogeneous-embedded systems.

5.3.4.2 Application and Benchmarks Evaluation

To further measure the performance of our extended heterogeneous MTAPI RTL, we also ported a set of real-applications and benchmarks. For these tests, we mainly focused on the performance gained by dispatching tasks or workloads onto the GPU-accelerators. Thus, the comparison was between the host-side OpenMP performance and the performance on GPUs.

5.3.4.2.1 LU Decomposition Performance The LU Decomposition is an implementation of the algorithm that calculates the solutions of a set of linear-equations. This is a challenging application, because of relatively large row and column dependencies for the input-matrix. The application is modified to use HIP and our heterogeneous MTAPI.

We analyze the details of the application performance by comparing the time for data-movement between the host and the device, kernel-execution time, and the overall CPU-time between the Carrizo and the TK1 boards. Moreover, we also compare the overall performance between the parallel host-side and execution on the GPU-accelerator.

As seen in the Figure 5.12, the overall execution time on the Carrizo board is much shorter than the Tegra K1 board. Furthermore, the difference is mainly due to the data-allocation and movement.

The time spent on kernel-execution for both boards are quite similar. For analysis, the fast data-transfer on the Carrizo board is the hUMA memory-architecture as

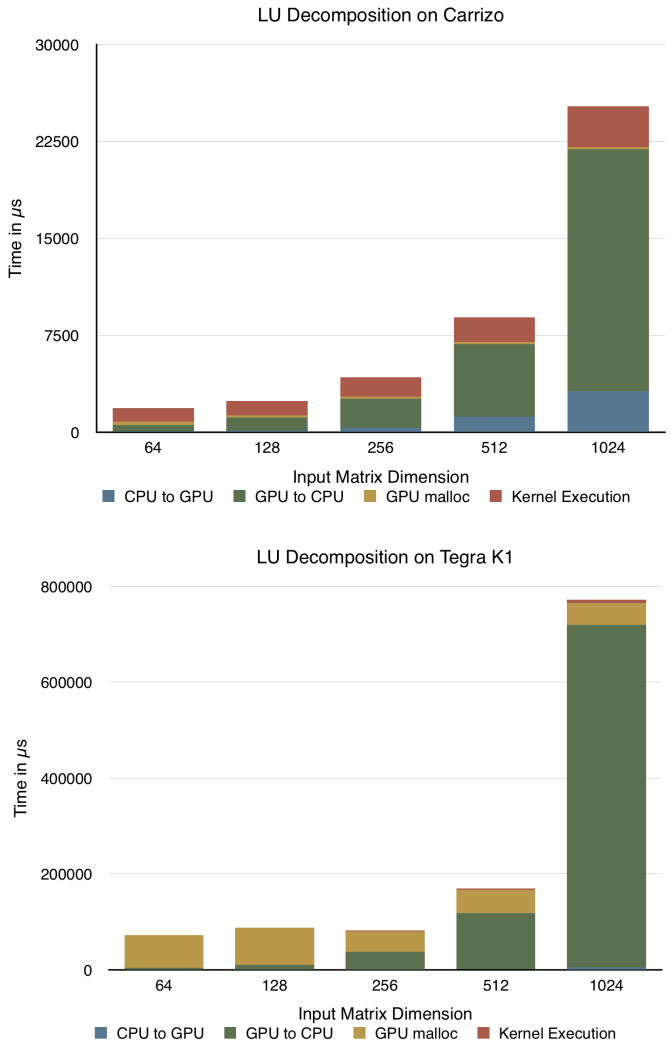

Figure 5.12: LU decomposition on Carrizo and Tegra K1 Boards, note the differences in the scale

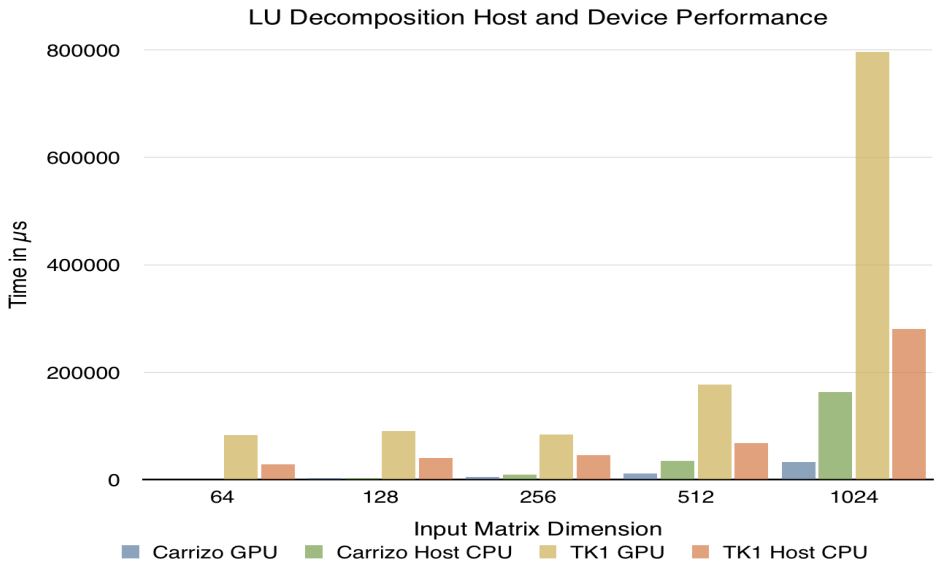

Figure 5.13: LU decomposition performance CPU vs GPU

discussed. In the hUMA memory-model, both of the host CPU and the GPU have the same view of the entire memory-space. Therefore, the data-movement for the Carrizo board only means to pass on the pointers between the CPU and the GPU.

However, on the NVidia TK1 board, the CPU and GPU units are logically two elements in the system, which have separate memory-space in the same physical-memory. Thus, the data transfer is enforced, before and after the kernel-execution, and those steps are the primary part of the execution. This test is a good example of the advantage in supporting bi-directional memory-coherence.

We further explored the speedup of execution on the host CPUs. We ran the OpenMP version of the LU Decomposition application on the host CPU side with four threads on both the Carrizo and the TK1 boards. The result is seen in Figure 5.13.

Because of the data-movement on the TK1 board, there are no performance gains on the GPU, compared to the TK1 CPU.

As the size of the input matrix increases, we see a larger performance advantage on the Carrizo GPU. The trend we observed is mainly due to the massively parallel-computation capacity provided by the GPU.

The data-movements for GPU systems are very costly and require careful optimization and performance-tuning. The HSA-compliant features provided by Carrizo minimize this concern. We achieved a better overall performance on the Carrizo board than the TK1 board, which highly ascribes to the HSA architecture.

By enabling HIP as the plugin, we extended our MTAPI runtime-library to support the heterogeneous-embedded systems that features GPUs. Furthermore, a much better performance was achieved on the HSA-compliant architectures.

Chapter 6

Map OpenMP onto MCA APIs

In this chapter, we introduce the project that maps OpenMP, the higher-level programming models, onto the MCA APIs. The mapping of OpenMP to MCA APIs provides higher-level programming-models while ensuring the portability of the applications using our solution stack.

6.1 Enhancing OpenMP Runtime Libraries with MCA API

In the embedded industry, the GNU compiler is the dominated compiler. We explored the suitability of MCA libraries with the GNU-compiler-based OpenMP runtime library, the libGOMP. We built an OpenMP runtime-library called *libEOMP* [68] where we mapped the essential resource-management functionality of the MCA

libraries to an OpenMP runtime-library implemented in OpenUH compiler [41]. OpenUH translates OpenMP directives and function calls into the parallel code for use with a custom runtime library (RTL). GCC's latest release provides support for OpenMP 4.5 on C/C++ compilers and FORTRAN, which is encouraging for multicore-embedded programmers to use OpenMP.

Compilers translate the high-level OpenMP pragma-based directives into an Intermediate-Representation (IR). The directives provide hints to the compiler to perform code-transformations, so that the serial-code can be converted into parallel-code. After translating the OpenMP constructs to C-like IR, the large part of the code is built into a separate runtime-library. This provides a set of high-level functions that are used to implement the OpenMP-constructs efficiently. The OpenMP runtime typically manages the parallel-execution by creating worker-threads, and managing thread pools and the synchronization among threads. An efficient runtime can also offer productive-scheduling, data-locality, and workload-balancing techniques.

In general purpose computation, the systems offer full OSes and multi-threading libraries that can be effectively utilized by the OpenMP runtime library. Unfortunately, most of the embedded systems do not provide these features since they are heavily customized to cater to specific applications. Hence, we use MRAPI as the translation layer for OpenMP, to target such complex systems. The work in this section has been published in [61, 68].

6.1.1 Memory Mapping

In the OpenMP runtime, some global data-structures must be maintained. For example, each team of nodes needs to keep a block of work-share to be assigned later. In this project, we extend the MRAPI shared-memory constructs to allocate thread-level shared-data, as seen in Listing 6.1

```
1  void *gomp_malloc (size_t size)
2  {
3    mrapi_shmem_attributes_t shm_attr;
4    shm_attr.use_malloc = MCA_TRUE;
5    mrapi_status_t mrapi_status;
6    mrapi_shmem_create_malloc(SHMEM_DATA_KEY,size,&shm_attr,&mrapi_status);
7    if (mrapi_status == MRAPI_SUCCESS) {
8      return shm_attr.mem_addr;
9    }
10   else
11     gomp_fatal("MRAPI failed memory allocation");
12 }
```

Listing 6.1: MRAPI memory extension

6.1.2 Synchronization Primitives

The synchronization-primitives of OpenMP-MCA runtime library has been mapped to MRAPI mutexes. This is to prevent critical data-races and manage accesses to the shared-data. Specifically, we use *mrapi_mutex_create* to create the mutex object upon initialization.

```
/* libGOMP Mutex Lock entry */
static inline void
gomp_mutex_lock (gomp_mutex_t *mutex)
{
  int oldval = 0;
  gomp_mutex_lock_slow(mutex, oldval);
}

/* MCA Mutex Lock entry */
static inline void
gomp_mrapi_mutex_lock (gomp_mrapi_mutex_t *mutex)
{
  mrapi_status_t mrapi_status;
  mrapi_status = MRAPI_SUCCESS;
  mrapi_mutex_lock(mutex->mutex_handle,&(mutex->mutex_key),MRAPI_TIMEOUT_INFINITE,
       &mrapi_status);
}
```

Listing 6.2: MRAPI mutex in libGOMP

Listing 6.2 illustrates the MRAPI mutex function used to enhance the original libGOMP runtime-library. MRAPI mutex maps the lock-operation for the target-system, thus, making this low-level operation portable across to various sets of systems that supportes MCA API.

6.1.3 Metadata Information

MRAPI metadata-constructs have been utilized in the OpenMP library. We used the MRAPI metadata-trees to retrieve the available number of processors online for node-management. Moreover, the information is used to serve the MRAPI nodes for

threads-management.

6.1.4 Evaluating GNU OpenMP with MRAPI

In this subsection, the performance of the extended GNU OpenMP library was measured with MCA API. A discussion on the performance follows.

Table 6.1: Relative overhead of OpenMP-MCA runtime library versus GNU OpenMP runtime

Directive	**4**	**8**	**12**	**16**	**20**	**24**
Parallel	0.98	1.04	0.73	0.98	0.98	1.03
For	1.00	1.10	1.10	1.01	1.31	1.49
Parallel_for	0.99	1.36	1.05	1.03	0.81	0.95
Barrier	0.90	0.93	1.13	0.90	1.48	1.32
Single	0.41	2.39	1.09	0.97	0.99	1.03
Critical	0.99	1.34	0.99	1.19	1.11	0.45
Reduction	0.98	0.97	1.00	0.94	1.07	1.01

We used EPCC [16] to evaluate the OpenMP-MCA RTL, and to measure overheads caused by adding MCA API. EPCC is a set of programs that measure the overhead of OpenMP directives. Besides, it is also used to evaluate OpenMP runtime-library implementations. Table 6.1 lists the performance for OpenMP-MCA library compared to the original GNU OpenMP library. In Table 6.1, we normalized the overhead of OpenMP-MCA RTL, to provide a relative performance. The smaller number indicates fewer overheads. OpenMP-MCA library does not incur major overhead, and it performs better for some OpenMP-constructs.

With different thread-pool sizes, the PARALLEL-construct performs better than libGOMP. And the overheads are slightly higher than libGOMP for the *for* construct.

Thus, the combined *Parallel for* construct has a similar overhead. Performance on the rest of the constructs is also comparable to libGOMP, with different sized thread-pools. We have achieved competitive performance for each of the OpenMP constructs on T4240RDB board, though OpenMP-MCA library still has more room to be optimized with a larger thread-pool.

Next, we want to ensure the correctness of our implementation. For this step, we used the OpenMP validation suite [67] to check if the enhancements made to the library did not cause a lose of functionality. The results from the validation suite also helped determine which implementation defects to fix.

We then evaluated our OpenMP-MCA RTL implementation using NAS OpenMP benchmarks [36]; the results are illustrated in Figure 6.1. The execution time was measured in seconds, and the performance comparison was between the OpenMP-MCA library and the proprietary GNU OpenMP library. The NAS benchmarks have several different data-sets available. Typically, size S and W, which are the smallest data-sets available, can be used to validate the correctness of the compiler being tested. The larger data-sets can be used to measure the performance of the compiler and the OpenMP runtime-library. In this project, we chose the size A of NAS benchmarks for our performance-measurement. As seen in the Figure 6.1, the execution time for the single thread is in seconds.

We illustrated the performance from single-thread to 24 threads, which is the maximum number of threads available on the T4 board. Besides the performance comparison, we also measured the speed-up rate of both libraries within the same graph.

As seen in Figure 6.1, the performance of OpenMP-MCA library is very comparable to the proprietary GNU OpenMP library. In CG, EP, and IS test cases, the OpenMP-MCA library showed better performance compared to the proprietary libGOMP. On the speed-up rate, most of the benchmarks perform well. On EP test case, both of the OpenMP libraries had a great speed-up rate. The rest of the benchmarks achieved speed-up around 15 with 24 threads.

The performance and speed-up rates seen in Figure 6.1 show that, by enhancing libGOMP library with MCA APIs, no significant-overhead was incurred. Also, it provides portable software stacks to target a large number of multicore-embedded platforms.

6.2 Mapping OpenMP Task Parallelism to MTAPI

We discuss our proposed light-weight, portable OpenMP library that maps its task-parallelism onto MTAPI. Task-parallelism is essential for embedded-products. The automotive-application is a good example; the state-of-the-art automobile chips need to handle different signals and executions from many various aspects. Those include electronic-control units for valves, fuel-injection, and sensors. Each of these operations can be classified as a task. Those tasks could be executed in parallel with the help of an adequate programming model, such as the OpenMP task-parallelism. Moreover, the vast potential for self-driven cars exacerbates the needs for task-parallelism. There are many other examples available in the embedded domain that requires the help of task-parallelism, including robotics and aircraft.

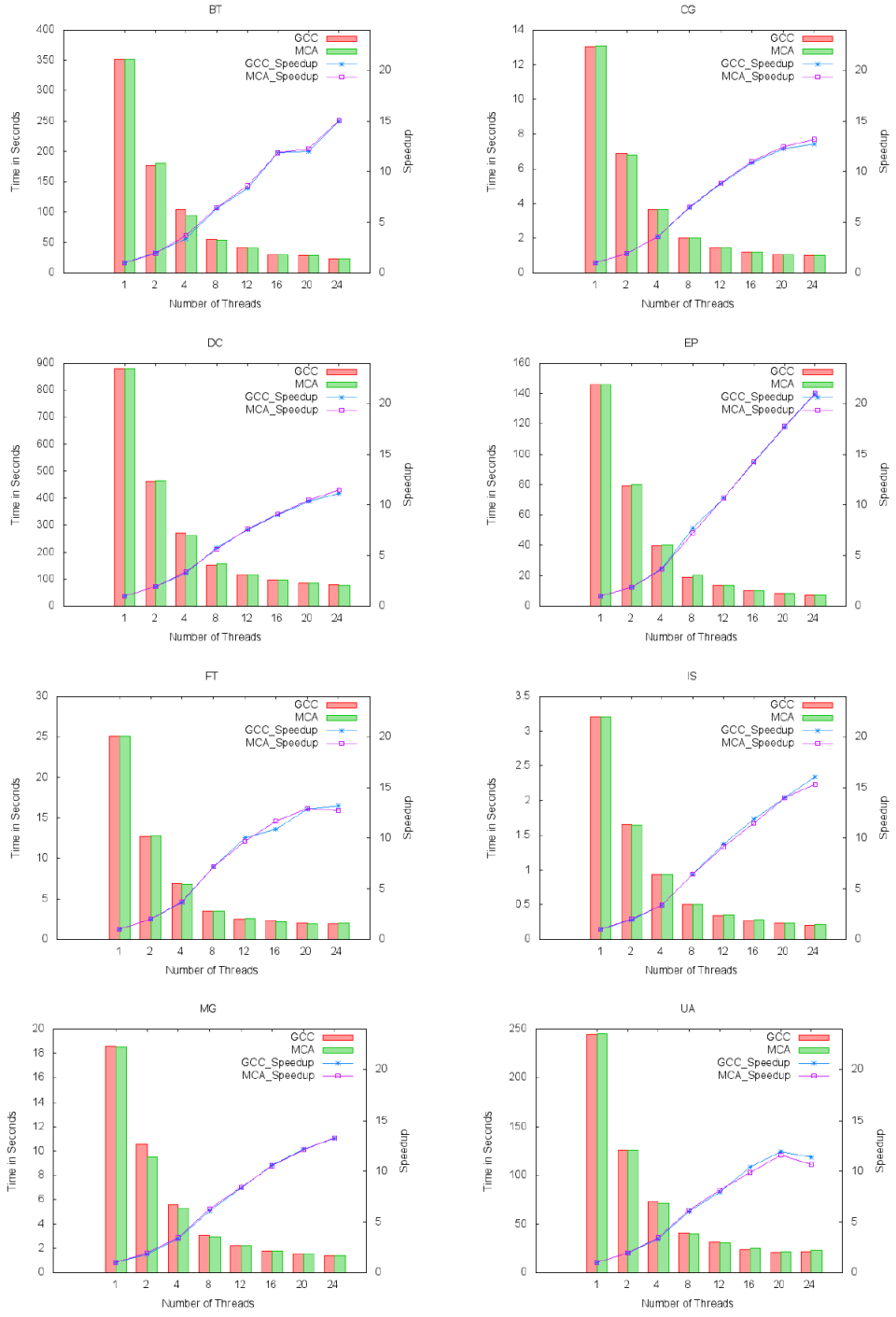

Figure 6.1: Evaluating OpenMP-MCA runtime library using NAS benchmarks

However, due to the impediments we discussed in the previous chapters, to map OpenMP task constructs onto embedded-devices is difficult. In addition, to build a portable implementation on devices with different architectures is even harder. Thus, in this section, we used the task management API (MTAPI) as the mapping layer of OpenMP task-constructs and achieved competitive performance. We have previously published this work in [62].

6.2.1 Overall Framework

In Subsection 6.2, we introduced our previous work that maps OpenMP task-parallelism support onto MTAPI library. In addition, MTAPI can take plug-ins to implement actions that execute on devices with different architectures. With the MTAPI library implemented in Section 5.3, further extended our solution stack onto heterogeneous platforms.

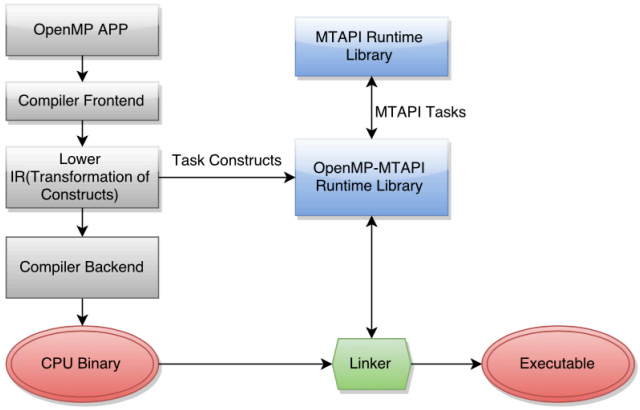

Figure 6.2: OpenMP-MTAPI solution diagram

Our proposed solution is to map the OpenMP constructs to MTAPI APIs. This is

to provide an elegant OpenMP programming interface and a comprehensive software stack for embedded systems. This work requires a broad learning of the OpenMP compilers and their runtime translations, as well as the environment-configurations. Figure 6.2 illustrates our framework. An OpenMP compiler front-end translates the OpenMP constructs into OpenMP-MTAPI library function calls. Therefore, the lower layer contains the transformation of the OpenMP application. The executable file is formed by linking OpenMP-MTAPI library and MTAPI library. During the execution time of the application, the OpenMP-MTAPI library takes the OpenMP task function-pointer and the task data-pointer to create a task. The OpenMP-MTAPI library then translates the tasks onto MTAPI task, and moves the control of execution onto the MTAPI library. After parallel execution, the MTAPI library sends the results back to the OpenMP-MTAPI library, thus accomplishing the execution of OpenMP-task.

6.2.2 Implementation

In this subsection, we give more implementation-level details of the work regarding the map of OpenMP task constructs onto MTAPI.

6.2.2.1 Parallel Construct

In the conventional OpenMP library, when the master-thread encounters an OpenMP parallel-region, a set of worker-threads are created and associated with a team. Thus, this is the "fork" of OpenMP's "fork-join" execution-model. After the creation of

worker-threads, the OpenMP library sets them into the state of *wait*. Later the library can wake them to working state. The overheads of the OpenMP-construct is reduced, since threads are not created multiple times.

In our enhanced OpenMP-MTAPI library, threads are not handled directly; which ensures portability. Instead, we used MTAPI to control the thread-management and workload-scheduling, to ensure the portability across different architectures and OSes. In our library, we initialized the default MTAPI node and created the default MTAPI action with the associated job handle when codes encountered the parallel-region. Therefore, the program can start MTAPI tasks without further initialization.

6.2.2.2 Task and Taskwait Constructs

OpenMP compilers translate each of the explicit tasks to a runtime-library function call with several parameters, including a function pointer, data frame, arguments, and dependencies. Inside the function, the tasks are created and put into the task queue for further execution. For *taskwait* construct, a typical OpenMP library specifies the descendant-tasks of the current task, and waits for their completion before moving to the next step.

We directly mapped OpenMP explicit tasks to MTAPI tasks in OpenMP - MTAPI library. This is done by sending the function-pointer and data-frame to the previously created MTAPI action, and starting the corresponding task-creations by calling *mtapi_task_create* routine. We also store the returned task-handle for further reference-created tasks. Besides, the optional OpenMP group IDs are mapped to the

MTAPI tasks. In the context of MTAPI, *task wait* utilizes the task-handle which is obtained from the creation of tasks, and then waits for the completion of the corresponding task. We assigned the OpenMP task-wait functionality onto MTAPI routines.

6.2.2.3 Taskgroup Construct

OpenMP 4.0 specification introduced the *taskgroup* construct, which provides a simplified task-synchronization mechanism. The taskgroup defines a structured region. All tasks created in the region, including their subtasks, belong to the same taskgroup. At the end of the taskgroup region, there is a synchronization point, which forces all tasks in this taskgroup to wait for completion.

MTAPI specification provides a set of routines for taskgroup management. When creating MTAPI tasks, programmers have options to associate tasks to a taskgroup. In our OpenMP runtime library implementation, when encountering OpenMP taskgroup region, we created an MTAPI task-group with the default group ID. Inside the group region while creating tasks, we specified which MTAPI taskgroup the tasks belonged to. We also mapped the exit-point of taskgroup onto the *mtapi_group_wait_all* routine. This was performed with the taskgroup handle obtained previously.

6.2.3 Performance Evaluation

In this subsection, we evaluated the implementation of our OpenMP-MTAPI runtime library. Our implementation achieved increased performance while ensuring

portability. We use the OpenMP Task micro-benchmarks [39] to demonstrate the overheads incurred by the OpenMP task-construct. We discuss the performance analysis by evaluating applications from OpenMP task benchmarks from Barcelona, the OpenMP Task Suite (BOTS) [26]. The experimental platform consists two E5520 CPUs, with 16 total threads for execution. The system provides sufficient resources to test over different numbers of threads. As mentioned earlier, we used the EMBB-MTAPI implementation for our prototype OpenMP-MTAPI runtime library.

6.2.3.1 OpenMP Task Micro-benchmark Measurement

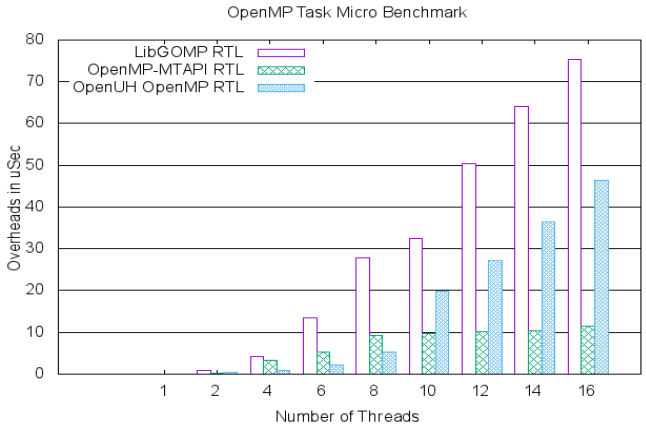

Figure 6.3: OpenMP task overheads measurement

To measure the performance of our translation, we first evaluated if the addition of an MTAPI-layer could result in any overhead. We used the OpenMP task micro-benchmark suite and compared the overhead of our OpenMP-MTAPI library against GCC OpenMP library. As a comparison, OpenUH-OpenMP runtime library was

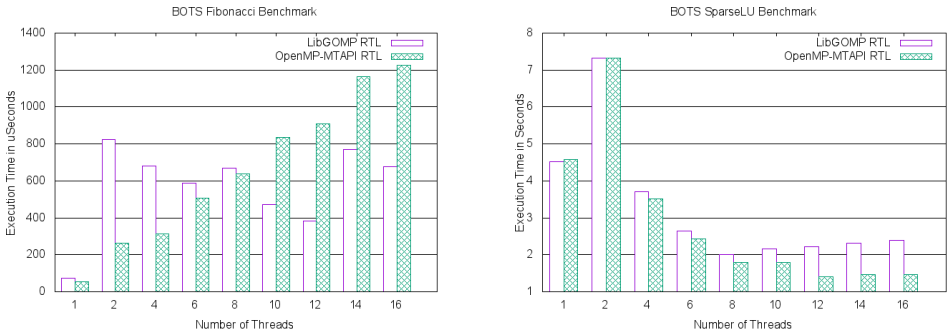

Figure 6.4: Evaluate OpenMP-MTAPI RTL with BOTS benchmarks

also tested [10]. The overhead in this scenario is the difference between parallel-execution time and the sequential-execution time over an identical section of code. During testing, the micro-benchmark conducted a large number of iterations, for both task-parallel executions and subsequential executions. Figure 6.3 illustrates the comparison of the overheads. In the graph, the Y-axis indicates the actual overhead of the OpenMP task-constructs in microseconds, for 1 to 16 threads. When the threads were less than 4, the OpenMP-MTAPI library performed similar to the GCC OpenMP library. However, the OpenMP-MTAPI library has significant advantages for larger numbers of threads. The performance of OpenUH library showed fewer overhead for less than eight threads; however, when the number of threads increased, the OpenMP-MTAPI library performed better.

Our runtime-library performs better in task overhead. Minimum overhead of tasks creation and scheduling over many threads is desired, when MTAPI is used on multicore-embedded devices.

In this subsection, we measured the performance of the OpenMP-MTAPI library using the BOTS suite. We chose Fibonacci and SparseLU for performance evaluation.

6.2.3.2 Fibonacci Benchmark

The Fibonacci benchmark uses a recursive task-parallelization method to compute the nth Fibonacci number. This benchmark features small computation loads, but heavy dependency and synchronization among the tasks. As seen in Figure 6.4a, the OpenMP-MTAPI library performs better than GCC OpenMP library with less than eight threads, while GCC OpenMP library has a distinct advantage with a larger number of threads. Our prototype implementation was further improved, especially for the synchronization strategies when using a large number of threads.

6.2.3.3 SparseLU Benchmark

The SparseLU benchmark computes an LU matrix factorization over sparse matrices. The matrix showed workload imbalance while using task-parallelism and dynamic scheduling lead to a better performance.

6.3 OpenMP-MTAPI for heterogeneous-embedded Systems

We designed and developed the heterogeneous MTAPI runtime library in Section 5.3. Moreover, to measure the performance of the extended-heterogeneous MTAPI library, we implemented benchmarks and applications. We achieved a relatively good performance on our targeted HSA-compliant Carrizo embedded-board. In this section, we introduce the OpenMP task constructs with our heterogeneous MTAPI

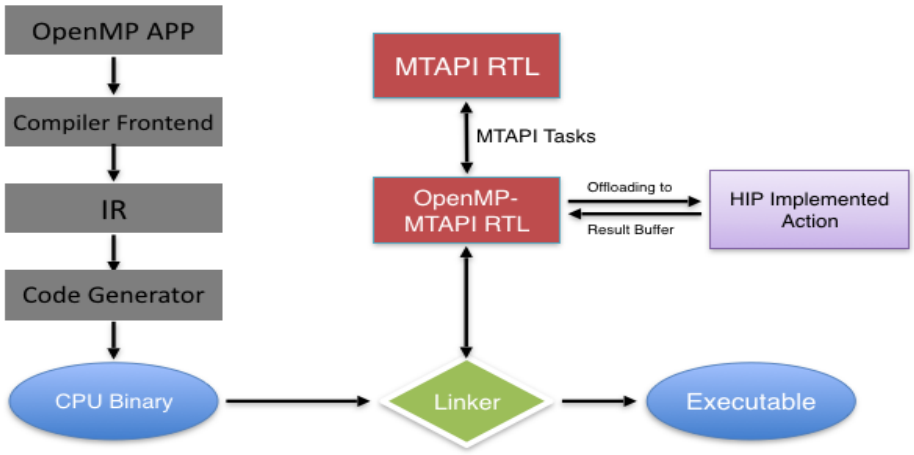

Figure 6.5: OpenMP-MTAPI with HIP plug-in

library, which enables the support of OpenMP task-construct on heterogeneous platforms.

6.3.1 OpenMP Task Construct with Heterogeneous MTAPI

Figure 6.5 shows our OpenMP-MTAPI flowchart, which takes HIP-implemented action on the MTAPI library. In our implementation, the functionality of communication and offloading tasks to HIP plugin are handled by the MTAPI library. The OpenMP-MTAPI runtime library is an intermediate layer that dispatches OpenMP tasks onto the MTAPI task queue. Also, it specifies the dependencies between tasks.

When an explicit task is created in the context of OpenMP task, the runtime library gathers the related task information and sends it to the task queue handled by the MTAPI library. MTAPI then schedules the task and arranges the tasks to

execute on the offloading device. The development platform is the same as discussed in Subsection 5.3.1.2, and we uses the same test-beds we adopted in Subsection 5.3.3, for both the AMD Carrizo, and Nvidia Tegra K1 boards.

6.3.2 Performance Measurement for Heterogeneous OpenMP-MTAPI

We measure the performance and efficiency of our solution-stack for heterogeneous platforms. The underlying heterogeneous-embedded platforms are the Carrizo and the Tegra K1 boards. We analyze each of the test cases by comparing the performance on the host and the accelerator. The time spent on data-transfer and kernel-execution were also tested.

We ported several test cases from Rodinia [19] and the Scalable Heterogeneous Computing (SHOC)[23] benchmark suite. Those benchmark suits are well-adopted for industry and academia. The main purpose is to measure the potential performance loss that is introduced by extending MTAPI onto heterogeneous platforms.

6.3.2.1 Grid Computation Evaluation

RTM8 was modified from an oil and gas industry application and ported to our solution stack. We use this application to evaluate the potential for speedups for such structured-grid applications. Moreover, the structure of the application features massive-parallel computation. We expect good performance for this application. The original application was programmed in FORTRAN. We use the single-thread

Table 6.2: RTM8 application

	Carrizo Fortran CPU	Carrizo Task GPU	Carrizo Speedup	TK1 Task GPU
PT Rate(millions/sec)	5.213	1195.23	229.26	987.65
FLOP Rate(Gflops)	0.3493	80.08	229.26	66.172

FORTRAN code as the base of the performance comparison.

6.3.2.1.1 RTM Application As seen in Table 6.2, a larger performance gain was achieved by executing the application on GPUs with our software stack. The performance on GPU was 229.26 times faster than that on single-thread host CPU. Since this application involves minimum data-transfer between host and device, we also obtained good performance on the TK1 board, about 82% of what we got from the Carrizo board. Certain applications that require massive-structured grid-operations are very suitable to execute on GPUs.

6.3.2.1.2 Hotspot Benchmark HotSpot is a widely used algorithm that estimates processor temperature. Each output cell in the computational grid represents the average temperature value an area on the chip. We modified the benchmark to work with heterogeneous MTAPI library and used by the OpenMP task construct. The performance data is illustrated in Table 6.3. In the table, we use TK1 to represent the performance of the Tegra K1 board and use CZ to account for the performance on Carrizo board. The time for each step was measured in *s*. The speed-up rate was calculated by dividing time of OpenMP on the host to the time of benchmark offloaded to GPU.

Table 6.3: HOTSPOT performance data

	CPU to GPU Data	GPU to CPU Data	GPU Data Allocation	Kernel Execution	Overall GPU Time	OpenMP CPU Time	Speedup
TK1_64	485	475	65365	325	66650	3220	0.05
TK1_512	3395	36903	44906	247	85451	29692	0.35
TK1_1024	11700	154722	49457	277	216156	81796	0.38
CZ_64	70	134	232	155	591	4437	7.51
CZ_512	1586	693	1360	157	3796	11544	3.04
CZ_1024	3595	5015	3452	169	12231	25338	2.07

By analyzing the performance table, the HotSpot benchmark achieved a much better performance on the Carrizo board, with an average speed-up by the factor of four. However, the benchmark ran very poorly on the Tegra K1 board. The data-transportation and device data-allocation used of the time for execution. As a result, though the kernel-execution time on both the Carrizo and Tegra boards were similar, the difference for overall performance were large. On the Carrizo board, the hUMA memory architecture and HSA architecture are enabled, which theoretically, has much smaller overheads to transfer data from the host to the device and back to the host. The same applies to data allocation. Better performance was achieved by adopting proper software tools and underlying heterogeneous-embedded system. Our solution stack can help programmers explore parallelism over devices across vendors, providing better performance and portability.

Chapter 7

Conclusion

7.1 Conclusion

This dissertation focused on developing a high-level, standard-based software solution stack for heterogeneous-embedded multicore systems. We have explored low-level Multicore Association standards that offer communication management, resource management, and task management APIs. Those APIs can help to manage data movement between cores, resource synchronization, and task parallelism on multicore systems. To abstract the software stack even further, we have adopted a high-level pragma-based standard approach, OpenMP, and translated OpenMP to the MCA APIs layer. To this end, embedded platforms that comply with MCA APIs can benefit from OpenMP. As we know, embedded systems are typically heterogeneous platforms consisting of either ARM + GPU, or ARM + DSP, or CPU + FPGA, or CPU + GPU; it is critical to extend the software stack so as to support these types

of systems. Which provide cores with features and varying functionality. However, without an efficient software stack, it is a challenge to tap their potential.

We propose a solution stack that maps OpenMP to heterogeneous-embedded systems by adding an HSA plug-in for the MTAPI library. This allows us to offload computations to HSA-compatible devices.

The overall contributions of this thesis are as follows:

- Portable software stack using OpenMP for heterogeneous multicore embedded systems.
- Extend MCA task management API to support heterogeneous systems across vendors.
- Evaluate the proposed solution stack on various embedded systems and show the effectiveness and portability of our solution stack.

7.2 Future Work

For future work, we would like to explore and evaluate our stack on more types of heterogeneous-embedded devices, which include bare-metal devices, DSP devices, and FPGA devices. Those devices are frequently used in embedded devices. However, the programming models for those devices are not available. We believe our solution stack will be helpful for system resource management and execution scheduling over the underlying devices.

OpenMP 4.5 has been released. The computation offloading to accelerators is better supported. Thus, it is desirable to explore the OpenMP *target* and *target data* constructs with our solution stack. We believe OpenMP will be the proper high-level programming model for future heterogeneous computing on embedded platforms.

Moreover, support for more programming interfaces in our solution stack can provide more options for the programmers. For the industries of the auto-driving car, medical imaging, and Virtual Reality, GPU is playing a more important role. The effort of introducing HIP to our solution stack is our first step to ease the programming on heterogeneous-embedded systems involving GPUs.

Bibliography

[1] ARM Mali-G71 GPU with HSA Support. https://www.arm.com/products/multimedia/mali-gpu/high-performance/mali-g71.php.

[2] Embedded multicore building blocks (embb). https://github.com/siemens/embb.

[3] Hsafoundation/cloc: Cl offline compiler : Compile opencl kernels to hsail. https://github.com/HSAFoundation/CLOC. (Accessed on 08/03/2016).

[4] Multicore Association Website. http://www.multicore-association.org.

[5] An open source implementation of the mcapi standard. https://bitbucket.org/hollisb/openmcapi/wiki/Home.

[6] Openmp 4.0 public review release candidate specifications. http://www.openmp.org/mp-documents/openmp-4.5.pdf.

[7] Polycore implementing a standard. http://polycoresoftware.com/news/implementing-a-standard.

[8] Siemens produces open-source code for multicore acceleration. http://www.techdesignforums.com/blog/2014/10/31/siemens-produces-open-source-code-multicore-acceleration/.

[9] Strided memory access on cpus, gpus, and mic — karl rupp. https://www.karlrupp.net/2016/02/strided-memory-access-on-cpus-gpus-and-mic/. (Accessed on 06/30/2016).

[10] C. Addison, J. LaGrone, L. Huang, and B. Chapman. Openmp 3.0 tasking implementation in openuh. In *Open64 Workshop at CGO*, volume 2009, 2009.

[11] A. Agbaria, D.-I. Kang, and K. Singh. Lmpi: Mpi for heterogeneous embedded distributed systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 8–pp. IEEE, 2006.

[12] P. Athanas, D. Pnevmatikatos, and N. Sklavos. *Embedded Systems Design with FPGAs*. Springer, 2013.

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[14] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 27–34. IEEE, 2010.

[15] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[16] J. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of the First European Workshop on OpenMP*, pages 99–105, 1999.

[17] D. Cederman, D. Hellström, J. Sherrill, G. Bloom, M. Patte, and M. Zulianello. Rtems smp for leon3/leon4 multi-processor devices. *Data Systems In Aerospace*, 2014.

[18] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[20] K.-T. Cheng and Y.-C. Wang. Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pages 1–4. IEEE, 2011.

[21] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload: Automating Code Migration to Heterogeneous Multicore Systems. In *Proceedings of HiPEAC '10*, pages 337–352. Springer-Verlag, 2010.

[22] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[23] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[24] T. Deakin and S. McIntosh-Smith. Gpu-stream: Benchmarking the achievable memory bandwidth of graphics processing units.

[25] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[26] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP'09*, pages 124–131. IEEE, 2009.

[27] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[28] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[29] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong. Device scaling limits of si mosfets and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, 2001.

[30] L. Gantel, M. Benkhelifa, F. Verdier, and F. Lemonnier. Mrapi implementation for heterogeneous reconfigurable systems-on-chip. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 239–239. IEEE, 2014.

[31] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[32] K. O. W. Group et al. The opencl specification. *version*, 1(29):8, 2008.

[33] J.-Z. He, W.-G. Chen, G.-R. Chen, W.-M. Zheng, Z.-Z. Tang, and H.-D. Ye. OpenMDSP: Extending Openmp to Program Multi-Core DSPs. *Journal of Computer Science and Technology*, 29(2):316–331, 2014.

[34] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing multiple starpu applications over heterogeneous machines: a supervised approach. *International Journal of High Performance Computing Applications*, 28(3):285–300, 2014.

[35] P. Jaaskelainen, C. S. de La Lama, P. Huerta, and J. H. Takala. Opencl-based design methodology for application-specific processors. In *SAMOS*, pages 223–230. IEEE, 2010.

[36] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[37] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[38] E. Konstantinidis and Y. Cotronis. A practical performance model for compute and memory bound gpu kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 651–658. IEEE, 2015.

[39] J. LaGrone, A. Aribuki, and B. Chapman. A set of microbenchmarks for measuring openmp task overheads. In *PDPTA*, volume 2, pages 594–600, 2011.

[40] J. Leskela, J. Nikula, and M. Salmela. Opencl embedded profile prototype in mobile device. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 279–284. IEEE, 2009.

[41] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. Openuh: An optimizing, portable openmp compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[42] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman. Early experiences with the openmp accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[43] P. Mahr, C. Lorchner, H. Ishebabi, and C. Bobda. Soc-mpi: A flexible message passing library for multiprocessor systems-on-chips. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*, pages 187–192. IEEE, 2008.

[44] A. Marongiu and L. Benini. An openmp compiler for efficient use of distributed scratchpad memory in mpsocs. *Computers, IEEE Transactions on*, 61(2):222–236, 2012.

[45] A. Marongiu, P. Burgio, and L. Benini. Supporting openmp on a multi-cluster embedded mpsoc. *Microprocessors and Microsystems*, 35(8):668–682, 2011.

[46] L. Matilainen, E. Salminen, T. Hamalainen, and M. Hannikainen. Multicore communications api (mcapi) implementation on an fpga multiprocessor. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 286–293. IEEE, 2011.

[47] A. Nicolas, H. Posadas, P. Peñil, and E. Villar. Automatic deployment of component-based embedded systems from uml/marte models using mcapi. *XXIX Conference on Design of Circuits and Integrated Systems,DCIS 2014.*, 2014.

[48] NVIDIA. Cuda C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[49] NVIDIA. The openacc specification, version 2.0, august 2013. *URL* ***http://www.openacc-standard.org/***, 2013.

[50] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 457–467. ACM, 2013.

[51] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez. Experiences with mobile processors for energy efficient hpc. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 464–468. EDA Consortium, 2013.

[52] A. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin. SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip. In *Proceedings of CASES ' 08*, pages 95–104. ACM, 2008.

[53] P. Rogers and A. C. FELLOW. Heterogeneous system architecture overview. In *Hot Chips*, 2013.

[54] F. Sainz, S. Mateo, V. Beltran, J. L. Bosque, X. Martorell, and E. Ayguadé. Extending ompss to support cuda and opencl in c, c++ and fortran applications. *Barcelona Supercomputing Center–Technical University of Catalonia, Computer Architecture Department, Tech. Rep*, 2014.

[55] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[56] S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, and C. Beame. Network file system (nfs) version 4 protocol. *Network*, 2003.

[57] K. Sollins. The tftp protocol (revision 2). `https://www.ietf.org/rfc/rfc1350.txt`, 1992.

[58] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault. Openmp on the low-power ti keystone ii arm/dsp system-on-chip. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 114–127. Springer, 2013.

[59] P. Sun, S. Chandrasekaran, and B. Chapman. Poster: High level programming model for heterogeneous mpsocs using industry standard apis. In *TECHCON 2014, in TECHFAIR 4*. SRC, September 2014.

[60] P. Sun, S. Chandrasekaran, and B. Chapman. Targeting heterogeneous socs using mcapi. In *TECHCON 2014, in the GRC Research Category Section 29.1*. SRC, September 2014.

[61] P. Sun, S. Chandrasekaran, and B. Chapman. Openmp-mca: Leveraging multiprocessor embedded systems using industry standards. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 679–688. IEEE, 2015.

[62] P. Sun, S. Chandrasekaran, S. Zhu, and B. Chapman. Deploying openmp task parallelism on multicore embedded systems with mca task apis. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 843–847. IEEE, 2015.

[63] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling a high-level directive-based programming model for gpgpus. In *Languages and Compilers for Parallel Computing*, pages 105–120. Springer, 2014.

[64] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[65] J. D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[66] S. Wallentowitz, P. Wagner, M. Tempelmeier, T. Wild, and A. Herkersdorf. Open tiled manycore system-on-chip. *arXiv preprint arXiv:1304.5081*, 2013.

[67] C. Wang, S. Chandrasekaran, and B. Chapman. An openmp 3.1 validation testsuite. In *OpenMP in a Heterogeneous World*, pages 237–249. Springer, 2012.

[68] C. Wang, S. Chandrasekaran, P. Sun, B. Chapman, and J. Holt. Portable mapping of openmp to multicore embedded systems using mca apis. In *ACM SIGPLAN Notices*, volume 48, pages 153–162. ACM, 2013.

[69] Y.-C. Wang, B. Donyanavard, and K.-T. T. Cheng. Energy-aware real-time face recognition system on mobile cpu-gpu platform. In *Trends and Topics in Computer Vision*, pages 411–422. Springer, 2012.

[70] M. Wu, W. Wu, N. Tai, H. Zhao, J. Fan, and N. Yuan. Research on openmp model of the parallel programming technology for homogeneous multicore dsp. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*, pages 921–924. IEEE, 2014.

[71] G. Xie, R. Li, X. Xiao, and Y. Chen. A high-performance dag task scheduling algorithm for heterogeneous networked embedded systems. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 1011–1016. IEEE, 2014.

[72] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman. Nas parallel benchmarks on gpgpus using a directive-based programming model. In *Intl. workshop on LCPC 2014*, 2014.

[73] C.-T. Yang, T.-C. Chang, H.-Y. Wang, W. C.-C. Chu, and C.-H. Chang. Performance comparison with openmp parallelization for multi-core systems. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 232–237. IEEE, 2011.

[74] H. Zhou and C. Liu. Task mapping in heterogeneous embedded systems for fast completion time. In *Proceedings of the 14th International Conference on Embedded Software*, page 22. ACM, 2014.