# 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

July 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. Special thanks to all my committee members: Dr. Edgar Gabriel, Dr. Shishir Shah, Dr. Jaspal Subhlok, and Dr. Sunita Chandrasekaran, for their time, insightful comments and help. Specifically, I am very grateful to Dr. Sunita Chandrasekaran for the valuable mentoring and guidance for my research and publications. That help is paramount to my Ph.D. Journey. 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.

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 moreover, 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

July 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 standards 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 contributions and the outlines of this proposal report.

## 1.1 Motivation

Programming embedded systems is a challenge, the use of low-level proprietary APIs makes the development process tedious and error-prone. As the trend that embedded systems are becoming more heterogeneous, for better performance and power efficiency. Hardware designers pack more and more specialized processors to the new embedded systems and SoCs, such as DSPs, FPGAs, and GPUs. Those hardware elements have different architectures and instruction sets themselves, making the software development kit more complicated, thus a steep learning curve follows.

Furthermore, to provide programming interface solution that could help programmers efficiently oversee and utilize the computation power from the hardware is remain a challenge, especially considering developing 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 among them. Without a portable and easy-adopted programming model, the goal cannot be achieved.

OpenMP[23] is considered to be a potential candidate to resolve most of the challenges we discussed above. OpenMP is a pragma-based programming model; it allows the software developers to parallelize a sequential code incrementally, providing data parallelism or task parallelism without the need to handle the lower-level threading libraries or OS level resources. Besides, OpenMP allows a compiler that does not support its features to ignore the directives, thus further guarantee the portability of the software products. With the release of OpenMP 4.5 specification[5], OpenMP expanded its coverage to heterogeneous architectures. The OpenMP *target* construct allows programs to dispatch part of the executions onto the accelerators. Early studies include [62, 44].

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 instance, most of the OpenMP compilers translate the pragmas into parallel codes with OS-level threading libraries. Cache coherence is expected by the

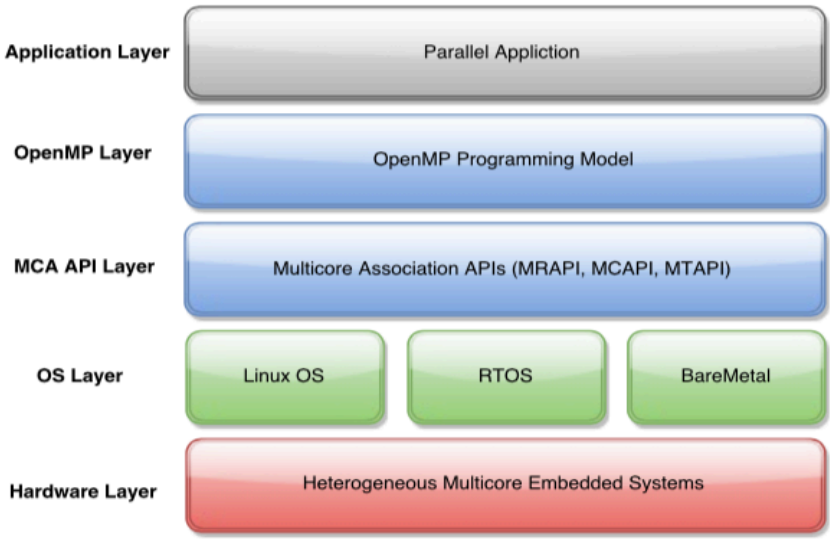


Figure 1.1: Solution Stack of Using OpenMP on MCA APIs

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

1.1 illustrate the solution stack we propose to resolve the challenges we discuss above. To bridge the gap, we choose to deploy the APIs defined by the Multicore Association (MCA)[3] as the translation layer of the OpenMP runtime library. MCA is a non-profit industry consortium that is formed by a group of leading semiconductor companies and academics, to define 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 [70, 64]. The results from our past work demonstrated that such an approach, which translating

OpenMP to MCA APIs, has a lot of potential both regarding programmability and portability.

In this dissertation, we propose a comprehensive solution stack and expandable platform, to help programmers better explore parallelism on the heterogeneous embedded platforms with the support of OpenMP, by using MCA APIs as the mapping layer, with suitable plug-ins and optimized scheduler, to provide portable OpenMP implementation across heterogeneous 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 abstract the 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. Including high-level programming model and the low-level industry standards that abstract the underlying system and software details. The following 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 the industry standards for abstraction. Chapter 4 provides our investigation and experiments to choose the proper plugin language for our solution stack, to 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 the broader support of target platforms, an extension to support on heterogeneous systems, and the discussion on the performance. Chapter 6 discusses the work we have done to map OpenMP, the higher level programming model, onto MCA APIs, to provide convenient programming model while guarantee the software portability. chapter 7 concludes this dissertation and discuss the future work we would work towards in the future.

# Chapter 2

# Background

In this chapter, we introduce the background of the dissertation work. In this work, we adopt OpenMP as the programming model, with the help of MCA APIs and HIP, to map on a various of homogeneous and heterogeneous systems. We also highlight some of the key features and usability of the related concepts, which would better help understand our dissertation work and the solution stack to the challenges. Noted the background of HIP and HSA will be introduced in section 5.3.

## 2.1 OpenMP

OpenMP [23] 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 serial 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.

The feature that OpenMP allow programmers gradually parallelize sequential codes, making it the best choice when developers want to explore the potential parallelism of existed legacy programs quickly. In the chapter of accomplished work of this report, we examined using MCA Resource Management APIs to manage the system and thread-level resource of the targeted embedded systems, to implement OpenMP data parallelism. Besides, we also managed to map OpenMP task construct onto MCA Task Management APIs to provide programmers with an easy-to-use programming interface of task parallelism for broad coverage of targeting 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 shown 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 forked 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 could be placed anywhere inside an OpenMP *parallel* construct, and explicit tasks are created when a thread encounters the *task* construct. Then these explicit tasks could 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.

```
1  int fib(int n)
2  {
3    int i, j;
4    if (n<2) return n;
5    else{
6         #pragma omp task shared(i)
7         i=fib(n-1);
8         #pragma omp task shared(j)
9         j=fib(n-2);
10        #pragma omp taskwait
11        return i+j;
12   }
13 }
14 int main(void){
15   #pragma omp parallel shared(n)
16   {
17     #pragma omp single
18     printf ("fib(%d) = %d\n", n, fib(n));
```

```
19   }
20 }
```

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 programmers on embedded systems has not been able to exploit this luxury, yet. Reasons behind sense are that typical embedded systems lack some of the features that are on general-purpose computers, which are essential for an OpenMP execution. OpenMP implementations usually rely heavily on threading libraries and operating system resources. However, embedded systems domain may lack such features; some embedded systems do not even have an OS such as bare-metal devices. It might seem simple enough just to offload most compute-intensive portions of the code to the accelerator or a specialized processor using conventional methods such as remote procedure calls, but alternate approaches such as dataflow-oriented have proved to be more efficient due to lesser application deadlocks; for devices like FPGAs[11]. The hurdle requires that programmers re-think ways to program codes for the type of specialized hardware resources an embedded device can offer.

As we can see, it remains a challenge to create an efficient programming (almost universal) interface for the embedded world without having substantial details of the hardware design and approaches to optimize an application for such platforms.

## 2.2 MCA Libraries

To address some of 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[3] 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 turbo 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 seeks to handle 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 virtually any number of cores, even each with a different instruction set, same or different OSes. Besides that, MRAPI could allow coordinated concurrent access to the systems resources by deploying the synchronization primitives for embedded systems that with limited hardware resources. The MRAPI could support a

varies 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 description.

#### 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 could 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 could allow users to manage the on-chip or off-chip shared memory directly, with assigned attributes. Unlike the Linux shared memory, which could only be accessed within one operating system's entity, the MRAPI shared memory could be accessed by different nodes running different OSes. The remote memory model enables the access of distinct memories. This model could be physically consecutive or not direct access, for the latter case, some other methods like DMA will need to be used to access to the remote memory. By providing unique API interfaces, MRAPI hides all

those 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 to the shared resources, to avert data race or race conditions.

#### 2.2.1.4 System Resource Metadata

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

MRAPI is an excellent candidate that could abstract the underlying both hardware and OS-level resources and provide a unique interface to the end programmers as well as the higher level language, such as OpenMP. In the dissertation work, 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 would prove that by adding MRAPI as an abstract layer, our OpenMP-MCA RTL will not incur extra overheads and could achieve very 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 [6][4] 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 provided 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 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 work. 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[7] as a part of the EMBB[2] library.

The Multicore Association(MCA) is formed by a group of leading semiconductor

companies and academies. Among several functionality, the Multicore Association API[3] 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 could let the software developers 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 consists of 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 could 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 the 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 weighted 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 could be 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 and achieved good performance results.

## 2.3 HIP and ROCm Stack

GPU has been promoted as the mostly used accelerators for both embedded systems and high-performance computing facilities. By using GPU as computation accelerators, programs could achieve significant power efficiency over the CPU alone; while the modern programming model for GPU makes it is much easier to port applications to GPU other than the other hardware accelerators, such as DSP or FPGAs. In this section, we introduce the background of the HIP and ROCm work from AMD, to provide an easy to adopt programming model and the fully open-source software stack to programmers on GPUs that targeting GPUs across vendors.

### 2.3.1 HIP Programming Model

CUDA has been widely used for GPU programming, a significant portion of libraries and applications utilizing GPU were built with CUDA. However, CUDA itself is a proprietary programming model of NVidia, which caused a series of drawbacks, including closed-source toolchain and application portability.

AMD recently announced its GPUOpen initiative that providing a comprehensive open-source software stack targeting programming on GPU. HIP, as part of the GPUOpen initiative, is intended to be adopted for bot new application development and port the existing GPU programs developed by CUDA. HIP itself is a very thin layer that directs the compilation and execution of the applications onto either NVidia path and AMD path. It makes it possible that by maintaining a single code space of C++ application with HIP routines, the programs could execute

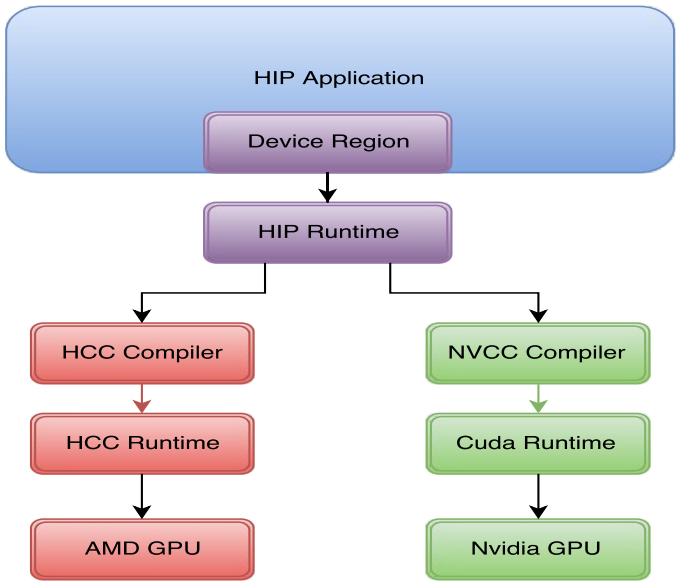


Figure 2.1: HIP on AMD and NV platforms

across platforms. Thus, the portability is firmly guaranteed. Besides, HIP supports the latest C++ language features, such templates, lambdas and much more.

As illustrated in 2.1, programs developed by HIP could be compiled and executed on platforms powered by both NVidia GPU cards 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 loose when comparing to the applications drawn up with CUDA on NVidia platforms. On the AMD path, HIP compiler will trigger HCC compiler as the C++ compiler for HIP applications, configure and set proper flags for compilation.

To aid the port of CUDA applications onto HIP, HIP provides a tool called

"hipify." It could perform a source to source translation over the existing CUDA applications onto HIP. The hipify tools are built with CLANG, the C/C++ front-end compiler of LLVM; thus the hipify tool code performs grammar analysis and makes sure the correctness of the translated CUDA applications could be compiled on both NVidia NVCC compiler and the AMD HCC compiler.

### 2.3.2 ROCm Software Stack

ROCm, the open source software stack targeting AMD GPUs 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 could compile both host code and device code offline

The ROCm system is currently supported for the AMD Fiji GPUs, which features the High Bandwidth Memory(HBM) on the device. In our initial experiments on the ROCm stack, we observed huge 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, at the rate of 512GB/sec.

With the release of ROCm by AMD, the GPU programmers have finally been able to see the who picture of GPU programming, from the top layer programming

model, runtime system, kernel driver, to the very lower ISA for device code. Thus, the programmers have the privilege to explore the full stack of the GPU apps, Whether for debugging or performance tuning, which was not possible with the closed-source toolchain. In the next paragraphs, we will discuss the technologies used in ROCm stack, which would be related to our decision to pick up on plugins for MTAPI.

**2.3.2.0.1 ROCm Linux Driver** The ROCm Linux driver initially targets on AMD APUs that fully compliant to the HSA specification, includes Kaveri desktop APUs and Carrizo embedded/notebook APUs; and later extended to discrete GPU support, includes the FIJI series GPUs and versions after that. 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 on the knowledge of HSA support for AMD APUs, and therefore extended to support dGPUs, such as FIJI series from AMD. The goal of ROCR runtime is to harness the power of GPUs to the applications by providing hardware-enabled dispatch queues for task offloading.

The ROCT Thunk Interface plays as the intermedia between the user-mode APIs and the ROCm kernel driver. The Thunk interface will be in charge to 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 efforts heavily depend on the existing work for HSA and supports the 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, a very active open-source compiler community. HCC implements several standards to enable programming on heterogeneous systems without touching proprietary tool-chain. By adopting the standards such as C++17, C++AMP, OpenMP and some others, the applications programmed with HCC could be 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, HSAIL, and the Lighting Compiler. The HSAIL backend comes with the efforts from HSA, and could generate fat binaries that execute on both HSA compliant devices and AMD dGPUs; while the Lighting Compiler is specifically designed and implemented for AMD GCN 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 some of these programming models onto some specific embedded systems, which shows the possibility of porting of these programming models onto embedded

systems.

### 3.1.1 Pragma-based Programming Model

OpenMP[23] is the *de-facto* programming model in shared memory computation, and has been extended to heterogeneous architectures with the release of OpenMP 4.0 specification. There are some efforts to mapping OpenMP onto the embedded domain. Our previous work[70] translates OpenMP to MCA Resources Management API, to support OpenMP on several Power based embedded systems from Freescale Semiconductor. [19] implemented OpenMP on a high-performance DSP MPSoC, with efficient memory management and software cache coherence.

TI[62] supports accelerator features from the OpenMP 4.0 standard. OpenMDSP[34], an extension of OpenMP, is 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 is not generic enough to be used for other systems. The authors in [72] conducts similar research. In[46], the authors discuss an OpenMP compiler for use of distributed scratchpad memory. 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 in[75]. Marongiu et al. [47] support OpenMP on a multi-cluster MPSoC embedded system. However, this work was not designed for portability, thus, would need a fair amount of efforts before it could be mapped to any other embedded devices. We see that there are many research activities that aiming to map OpenMP onto embedded systems, furthermore making OpenMP a

suitable candidate for programming parallel applications for embedded systems.

OpenACC[51] is another well-adopted pragma-based programming model to program to heterogeneous platforms consisting of accelerators such as GPUs, related work includes[74, 65]. 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[32] is the dominated message-passing programming model for distributed-memory environment computing. However, MPI is too feature rich for embedded systems. Some projects[45, 10] implemented subsets of MPI standards to support embedded distributed systems, but obviously, the portability issues is still a big hurdle for MPI to support a larger range of embedded systems.

OpenCL[33] 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 device, such as[42, 36]. OpenCL is still too low level, and performance portability is a big issue when reusing OpenCL programs for different architectures[29].

OmpSs[26] programming model helps program on clusters of GPUs, and it has been extended to use CUDA and OpenCL recently[56].

StarPU[12] is designed for heterogeneous platforms, aiming to provide a runtime 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.

We see that there are some models, libraries and language extensions to programming accelerators and heterogeneous systems[39] but their adaptability and suitability for embedded platforms are questionable for the several reasons we have previously discussed. In our work 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 could be considered as an appropriate programming model for embedded systems. The OpenMP *parallel for* construct could conduct data parallelism in an efficient manner, the *task* and *taskgroup* construct could allow programmers to quickly explore task parallelism for irregular algorithms, and the newly added *target* construct brings significant potential to support various types accelerators for offloading computation. However, the lack of resource, different architectures, and the low-power requirements are still preventing the broad support of OpenMP on embedded systems. With the proposed solution framework in this report, we could provide the embedded programmers an easy-to-adopt parallel programming model while ensuring a broad programming support for targeting embedded systems.

## 3.2 Standards for Programming Multicore Embedded Systems

In this section, we discuss the standards or libraries to abstract various embedded architectures.

### 3.2.1 Tools and Libraries for Multicore Embedded Systems

Developing software products from vendors for embedded systems typically require implementations at the lower-level vendor-provided APIs level; these do not follow any standards or specifications, thus making the learning curve steep while no software portability is guaranteed. Adopt industry standards to abstract the challenges is crucial, thus drawing much attention from the industry and university researchers. The Multicore Association (MCA)[3] was founded by a group of leading semiconductor companies and universities, that provides a set of APIs for abstracting the low-level details of embedded software development, and ease the efforts to resolve resource concurrency, communication and task parallelism.

Embedded systems are various in organizations and architectures. A particular programming challenge with such a system is working close to the hardware. Due to the necessity of tackling low-level details, C is the most used language for embedded application developments. Besides, vendors typically offer Software Development Kits (SDKs), specifically fit their own devices; further makes programming embedded devices not portable and error-prone.

Language extensions have been proposed[54][22] to abstract the low-level operations for a certain set of functionality on embedded systems. OpenCL and CUDA work closer to the platform. OpenCL[61] is a programming language and framework allow programming on heterogeneous platforms[13, 63] . However, the model is low-level and not easy to use. CUDA[50] is a popular programming model for GPU programming, but this is proprietary and limited to NVIDIA devices. There are several efforts that use CUDA to program NVIDIA devices such as Tegra mobile processor, for embedded applications, such efforts include [71, 21, 53].

### 3.2.2 MCA APIs

Many different facets of MCA are currently explored such as[38] explores MCAPI abstraction also for hardware components to allow portability between software and hardware. MCAPI was implemented on FPGA, and suitability of MCAPI for MP-SoC is discussed in[48]. A proof-of-concept of MRAPI on a Xilinx Virtex-5 FPGA has been applied in[31]. 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[49].

With the release of MTAPI specification, we see more research and industry efforts towards adopting 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, as the reference implementation for MTAPI specification. We use the *EMBB* MTAPI implementation

in our OpenMP-MTAPI runtime library, which is discussed in Section 5. European Space Agency (ESA) builds their MTAPI implementation[18] for effortless configuration of SMP systems in their space products. Stefan et al.[68] provide a baseline implementation of MTAPI for their research on open tiled many-core SoC.

### 3.2.3 HSA Standards

HSA[55] has been introduced by AMD, ARM, and other semiconductor companies; it aims to provide hardware and software solutions to reduce disjoint memory operations for heterogeneous architectures. Heterogeneous System Architecture (HSA)[55] is a consortium lead by AMD and ARM; it aims to provide hardware and software solutions to reduce disjoint memory operations for heterogeneous architectures, promoting GPUs as the first class of co-processors, and providing system support for other embedded computing devices. Unlike MCA APIs, the HSA expects a system level support for their specifications, for both hardware and software design. Besides, there are some programming languages designed for other devices such as FPGAs[14] and several others. CAPH[58] implements stream processing applications on FPGAS.

HSA-complaint devices have been implemented on AMD Kaveri series desktop APUs and Carrizo series embedded/laptop APUs. Besides, the Bifrost Mali GPUs[1] also has the full support of the HSA specifications. The Mali series GPUs has been widely used in smartphones and other mobile devices, over 750 million units of Mali GPUs were shipped in 2015 alone; broader research and implementations will be

expected to HSA on mobile devices.

There are also several academic activities were built on top of the HSA standards. [52] develops to mitigate the coherence bandwidth effects of GPU memory requests, on top of the CPU-GPU hardware coherence commitment from HSA standards.

To develop applications using HSA, the programmers could either adopt the HSA runtime library directly. Also, there are several higher level programming tools has enabled easier programming on HSA platforms. CLOC compiler and SNACK allow users to write OpenCL-like kernels for HSA application; the kernel function is treated as a callable host-side function with required launching parameters. HIP and ROCm stack, discussed in section 2.3, is another alternative programming models for HSA compliant systems. In this dissertation, we adopt HIP as the plugin for MTAPI, thus mapping computations onto heterogeneous embedded systems.

## 3.3 HIP for GPU Programming

AMD announced its GPUOpen project in late 2015. As one of the two components of GPUOpen, professional computing provides a comprehensive open-source software stack for GPU programming. The tools provided could aid with the tedious programming experiences for traditional GPU programming, and enable the possibility to trace problems down through the open-source stack or tune for better performance. There are several components in the stack are associated this effort, which was introduced in section 2.3.

## 3.4 Task Scheduling over Heterogeneous Systems

In general, the task scheduling problem is known to be NP-Complete[67]. However, due to the crucial role of scheduling heuristics plays in performance, there are intensive studies and various scheduling heuristics has been conducted towards how to schedule efficiently tasks over resources and the order of them. To resolve the challenge to task scheduling over heterogeneous embedded systems, some algorithms are proposed. The method that uses directed acyclic graphs(DAGs) to represent tasks and edge represent the dependencies has been widely adopted to schedule tasks over heterogeneous architectures; research includes [57, 66, 15]. [16] compares eleven static heuristics for mapping tasks over heterogeneous distributed systems. However, these work are not specifically target embedded systems. Thus, a set of the main considerations is missing for considerations from the scheduling heuristics, such as limited hardware resources and various of system architectures.

Some work targets to resolve the challenge to task scheduling over heterogeneous embedded systems. [73] proposes a heterogeneous tasks scheduling algorithm based on DAG algorithm over networked embedded systems, this work shows a significant potential of apply DAG heuristics onto heterogeneous embedded systems if taking the characteristics and features of the target platform into consideration of the algorithm design.

Some papers discuss the efforts to extends the StarPU[12] runtime library for better performance. [35] composing multiple StarPU applications over heterogeneous systems by introducing a hypervisor that automatically expands or shrinks contexts

using feedback from the runtime system. [76] proposes a theoretical framework and three practical mapping algorithms, their proposed algorithms can achieve up to 30% faster completion time. However, this work only considers the scenario of task scheduling between CPU and GPUs.

In the dissertation, we have done initial exploration of performance tuning of the MTAPI scheduler to serve the heterogeneous computation environment better, especially with GPU plugins.

# Chapter 4

# GPU Plugin

GPUs have been heavily used for general purpose computing in the recent years, for their massive parallel architecture and efficient programming model. Some applications could achieve tens or hundreds of speed ups when ported to GPUs. However, most of the applications available for GPUs are using the proprietary CUDA language and its infrastructure in HPC and workstation. We have also observed the similar trend for the embedded system domain; as the applications of GPU based embedded system in the self-driving, deep learning, and virtual reality has been rapidly growing.

As we discussed in chapter 1, MTAPI from the Multicore Association standard APIs could be extended onto heterogeneous architectures by adopting plugins. Thus, the choice of the plugin languages is critical to the dissertation work. In the dissertation, we would like to enhance the solution stack we proposed by covering those embedded platforms equipped with GPUs. To compare the programming languages

for the state-of-the-art GPU architectures, we conducted an investigation to HIP, CUDA, and OpenCL, and provide a companion regarding their programming model, memory model and the performance on various GPU platforms.

## 4.1 Open Source Programming on GPUs

There are more and more GPU-accelerated applications have been developed in the state-of-the-art HPC and workstations. However, a big portion of those apps was created with the proprietary CUDA language and its infrastructure. One of the problems with using proprietary software is that it is almost always tightly controlled by its owner. Also, the toolchain is mostly closed-source and will not be accessible to the programmers with complete pictures. With CUDA, the hardware options are limited to NVidia only, which causes less competition and severe software portability problem.

HIP and its tools, proposed and developed by AMD, allows developers to convert CUDA code to common C++ on both NVidia and AMD platforms. This new Heterogeneous-compute Interface for Portability, or HIP, is an instrument that provides GPU programmers with more choice in hardware and development tools. Besides, the HIP provides a clang based tool called "hipify," to help developers quickly transform their existing CUDA based codes onto HIP, without much manual modification. The whole software stack(ROCm) on the AMD path for professional computing is completely open-sourced, which means the programmers could easily trace their source code down through the stack and tune for better performance as needed.

OpenCL is a cross-platform programming language for heterogeneous platforms, not limited to GPU, but also for the other architectures, including CPU, DSP, and CELL. Both HIP and OpenCL could be used to write programs for GPUs across vendors. Thus, it is desirable to compare HIP and OpenCL in this chapter as well, to choose the proper heterogeneous plugins for the MTAPI RTL.

Overall, HIP has certain advantages when comparing to OpenCL regarding programming on GPUs. Firstly, with HIP, the programmers could write both host and devices C++ code in the source files with support for the latest C++17 features, such as templates, lambdas or classes. Also, unlike OpenCL, which requires compilation of the kernel at runtime, HIP could compile both of the host code and the device kernel code offline with HCC, and create a single fat binary for execution. Secondly, the grammar of HIP is more straightforward than OpenCL, which means programmers are not supposed to have a steep learning curve to move onto HIP from their familiar programming languages. Also, HIP plays as a part of the ROCm, an open source software stack by AMD; the programmers have significantly more flexibility to track problems down or tune for better performance.

We believe it is crucial to developers by providing industry standards and their open source implementation, to hide the lower level systems details and the varies of different targeting architectures. For GPU programming, it is of particular importance, as the heavy use of proprietary CUDA language will potentially cause monopoly of the GPU market, and weaken the competition for improvement.

## 4.2 Performance Evaluation of HIP

Before we decide which plugin to adopt in our solution stack, we tested the performance of HIP over general computing GPU platforms from both AMD and NVidia. The motivation is to verify if the HIP tool could achieve good performance, and this is the prerequisite to be adopted in our solution stack for heterogeneous computing. Besides, it would be interesting to show the potential to use HIP as the programming tool on GPUs across vendors, which is another feature that desired in my dissertation.

We have configured test platforms that from both AMD and NVidia. The AMD test bed features a Fiji Nano GPU card and the NVidia test equipped with one GTX Titan X GPU. Those GPUs are all high-end consumer grade GPUs and are very powerful for general computing. The main features for AMD Fiji Nano GPU cards are listed as follows:

- Featured 4GB HBM (High Bandwidth Memory)
- Global Memory Bandwidth is 512GB/sec
- 4096 Streaming Processors
- 8192 GFLOPS Computation Capacity
- TDP: 175W

Besides, the main features of Titan X GPU card as follows:

- Featured 12GB DDR5 Global Memory
- Global Memory Bandwidth is 336GB/sec
- 3072 CUDA Cores
- 6144 GFLOPS Computation Capacity
- TDP: 250W

We could see the hardware capacity for both of the GPU cards are on par, and the major difference is the type and size of the global memory. FIJI Nano GPU is one of the first GPUs that features the HBM memory, which could deliver very high memory bandwidth rate at 512GB. However, due to the limitation of the new HBM manufacture technology, there are only 4GB HBM on the device, which is quite restricted for general purpose computing. While the Titan X GPU features much larger 12GB DDR5 Memory, but smaller memory bandwidth comparing to FIJI Nano GPU. Overall, that two card would be good candidates for our test purpose.

We use a benchmark which simulates n-body algorithm and ported to HIP. We executed on both AMD FIJI NANO GPU and NVidia Titan X GPU, aiming to illustrate by using HIP as the programming model, the application could be executed on GPUs from both NVidia and AMD. Also, on NVidia GPU, by implementing the application in HIP, the application would perform as good as that implemented in CUDA.

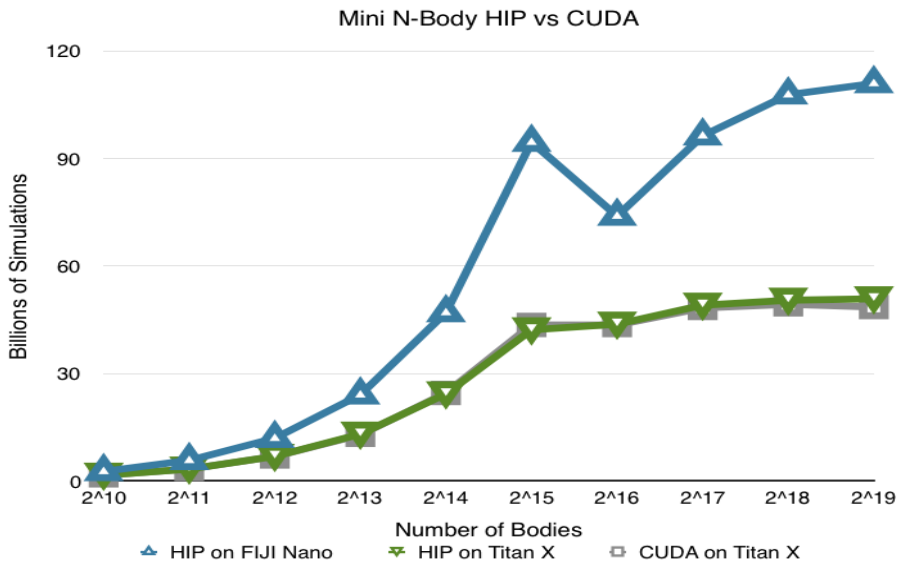


Figure 4.1: Mini N-Body Simulation Performance Comparision

As shown in Figure 4.1, we see the higher computation capacity and global memory bandwidth helps FIJI Nano achieves better performance. The overall performance on the FIJI Nano platform is much greater than that on the Titan X GPU card. As the size of simulation gets larger, the difference on performance gets larger. Moreover, as expected, the benchmark implemented in HIP and CUDA achieves very similar performance on the Titan X platform. The reason is on the NVidia platforms; HIP only plays as a very thin layer that translates the HIP routines back to CUDA routines.

By observing the benchmark we have conducted in this section, we could verify that HIP has a potential to be the programming model for GPU devices that across vendors. Thus we decide to develop HIP as the plugin target for our heterogeneous MTAPI implementation, and further, extend the use of HIP onto heterogeneous

embedded systems.

# Chapter 5

# MCA API Work

In this chapter, we firstly introduce the work we have done towards extending and enriching the industry standard APIs defined by the Multicore Association, the MCAPI, and MRAPI. We study and implemented MCAPI onto a heterogeneous embedded board, and extended MRAPI with more comprehensive thread-level resource management capacity. Then we introduce the work that extends the Multicore Association Task API (MTAPI) onto heterogeneous embedded systems by defining external plugins that allowing task offloading onto devices. For this dissertation work, we adopted the HIP programming language as the plugin to MTAPI, making applications written in MTAPI could be executed on embedded systems that features GPUs from both of the existing GPU vendors.

In the next chapter, we will talk about the work we have done on top of low-level MCA API work. Then we talk about the work we have done that portable map OpenMP onto embedded systems by use MCA APIs as the mapping layer.

## 5.1 Implement MCAPI on Heterogeneous Platform

In this project, 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 API i.e. MulitCore Association (MCA) API; that specifies essential application-level semantics for communication, synchronization, resource management and task management capabilities among several others.

For the task accomplished in this dissertation, we have explored the multicore communication APIs (MCAPI) that is 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 that is establishing communication mechanism between the P4080 multicore processor and the Data Path Acceleration Architecture (DPAA) incorporating acceleration for *Security Engine*(SEC 4.0) and *Pattern Matching Engine*(PME) accelerators. In this work, we study and managed to port MCAPI reference implementation onto a heterogeneous architecture, the P4080DS board from Freescale. We mapped the

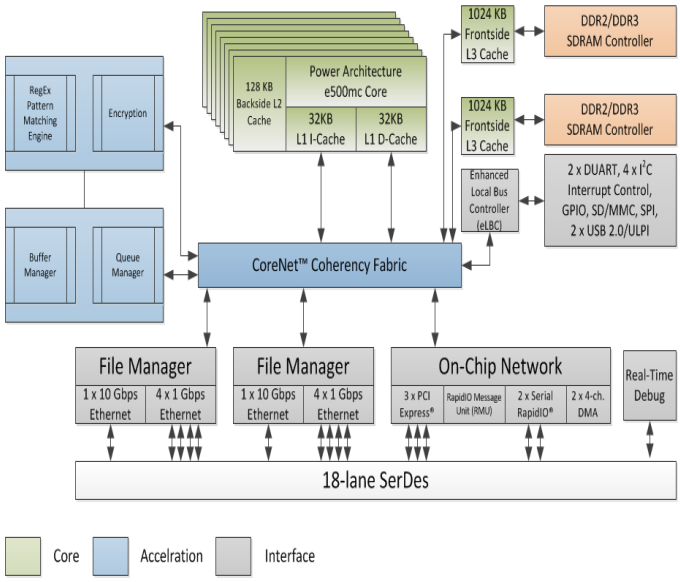


Figure 5.1: P4080 Block Diagram

### 5.1.1 Target Platform

The P4080 development system is a high-performance computing, evaluation, and development platform supporting the P4080 power architecture processor. Figure 5.1 shows the preliminary 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 32KB of instruction and data cache per core, 128KB 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 (DPAA) that offload various tasks from the e500mc, including routine packet handling, security algorithm calculation and pattern matching. The P4080 processor could be 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, long-term evolution (LTE) and general-purpose embedded computing systems in the networking, telecom/datacom, wireless infrastructure, military and aerospace markets.

#### 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*(SEC 4.0)
- *Pattern Matching Engine*(PME 4.0)

DPAA plays an important role in addressing critical performance problems in high

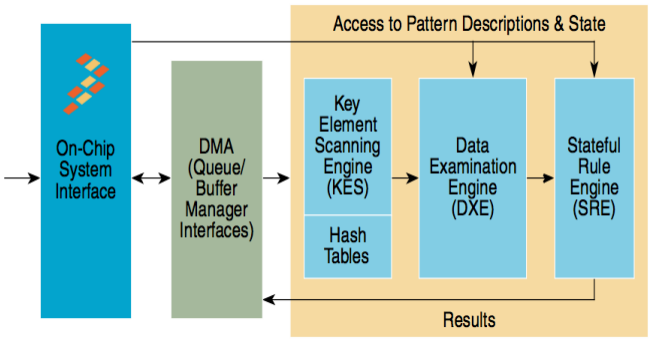


Figure 5.2: P4080 PME

speed networking I/O. It provides a bundle of user space APIs to be called by the end user to customise accelerator parameters and configurations.

#### 5.1.1.3 Pattern Matching Engine(PME)

The PME could provide hardware acceleration for regular expression scanning, scan across streamed data and find 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 for 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 for our work. We intend to abstract 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 that is also a bare metal hardware accelerator. This hardware accelerator does not offer shared memory support with the host processors, the power processor cores (e500mc cores) for P4080.

As a result, the data movement between the host and the accelerators need to be handled via a *DMA channel* explicitly. Thus, we will be deploying 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

We firstly studied functionalities and the prototype implementation of MCAPI. As mentioned earlier, MCAPI is an industry-standard API for inter-core communication within a loosely coupled distributed embedded SOC. It can be treated somehow like MPI, if lighter and designed for the embedded platforms. Nodes are a fundamental

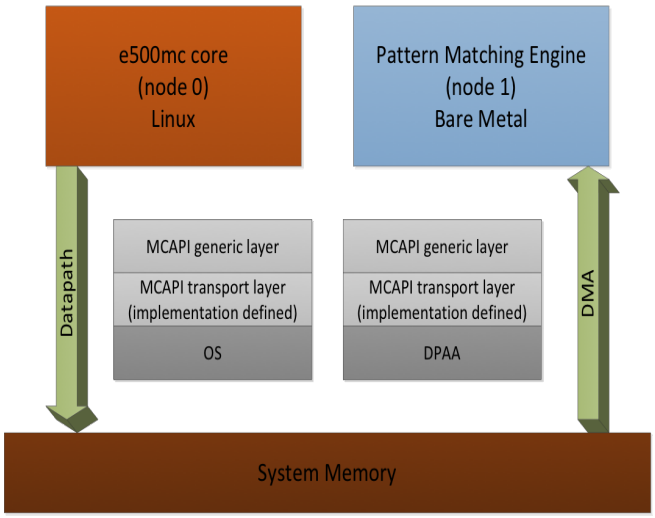


Figure 5.3: Solution Stack

concept in the MCA interfaces that map to an independent thread of control, such as a process, thread, or processor. In our implementation phase, using P4080, we set up the PME as a "node" in MCA and e500mc processors as the other node. Our solution stack is shown on figure 5.3.

We, therefore, explored the MCAPI transportation layer that will serve as the communication layer for PME. We modified the 'create end point' function to cater to the PME platform. We also modified the mcapi_trans_connect_pktchan and mcapi_trans_open_pktcha to build the MCAPI transportation channel for PME, that will appear on top of the DMA channel for PME package transportations.

For the MCAPI message passing, we came across the Pattern Matcher Control Interface (PMCI) library, a C interface to send and receive pattern matcher

control commands to PME. We found that this functionality could be partly abstracted by utilizing the MCAPI massage mechanism, for e.g. mcapi_msg_send and mcapi_msg_recv. To implement PMCI support within MCAPI transportation layer, we added 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 pmci_open has been mapped to the MCAPI transportation layer initialization subroutine. The pmci_close has been included in the MCAPI transportation finalize subroutine. The pmci_read and pmci_write have been mapped into the MCAPI message receive and send subroutines while the pmci_set option has been incorporated into the MCAPI node and endpoint initializations.

As seen on figure 5.4, we set the Power processor e500mc as one node with a dedicated endpoint, and the PME as the other node in MCAPI 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

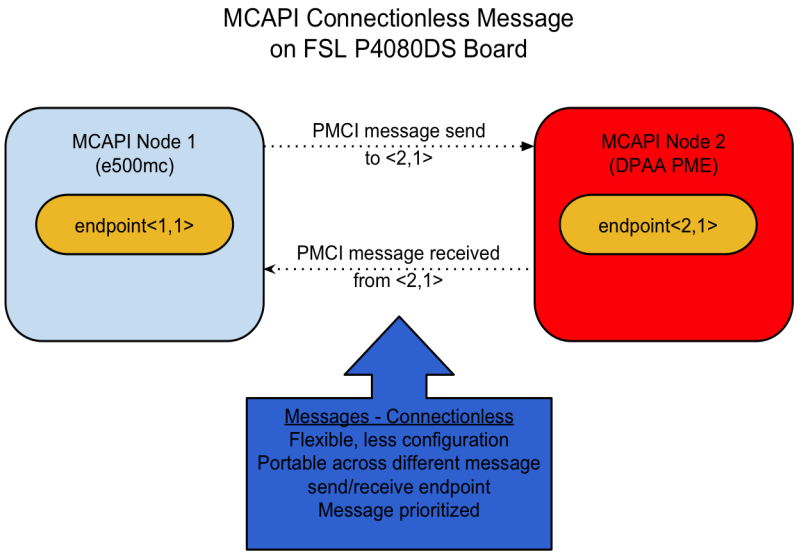


Figure 5.4: MCAPI on PME

build the connections between the nodes beforehand. This enables the flexibilities to send or receive messages between nodes and endpoints, as well as the possibilities to communicate among multiple endpoints simultaneously. Another important feature provided by MCAPI is the capacity to manage prioritized messages, which has potential to be utilized in our future work in the OpenMP runtime to handle time sensitive scenarios.

## 5.2 Multicore Resource Management API extension

MRAPI naturally 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 too 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 of the processes would run their private address space; thus one process could not access the other process's data unless utilizing an IPC method. Unlike processes, threads are lightweight. The latter has an advantage due to its lower cost of creation and the ability to exchange large data structures simply by passing pointers rather than copying. Thread synchronization does come with a penalty, and it is a challenge to create a thread-safe multithreaded program. OpenMP designers provide an easy method to write a multithreaded application without requiring the programmer to worry about creating, synchronizing and destroying threads. So in our work, we

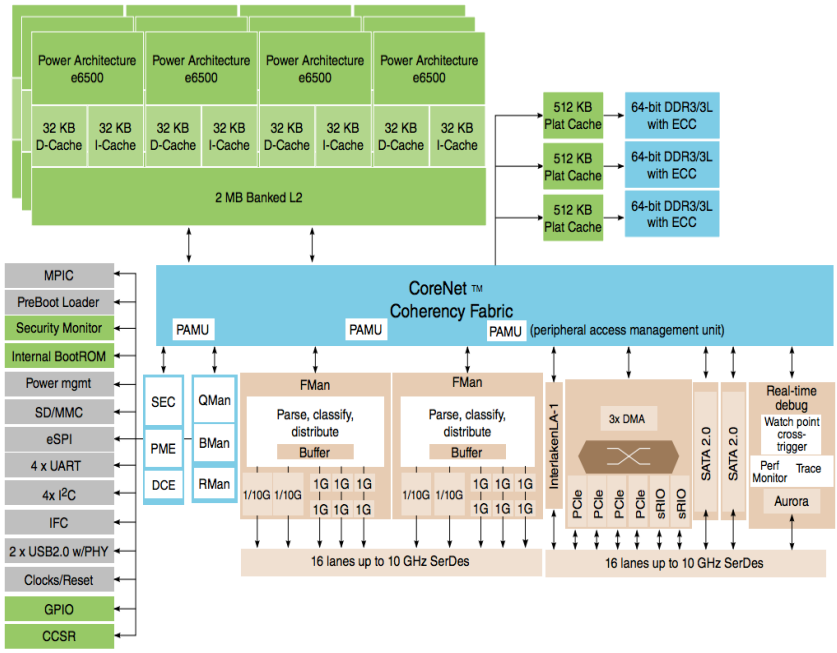


Figure 5.5: T4240RDB Block Diagram

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

#### 5.2.1.1 T4 Processor

The T4240RDB platform features twenty-four virtual threads from twelve PowerPC e6500 cores, running at 1.8GHz and providing the rich I/O capabilities. The Freescale T4 processor family is commonly used in networking productions like 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 AltiVec SIMD processing units are clustered in the T4240RDB board, manufactured in a 28nm 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, therefore, could be 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 the three groups in the T4240RDB board are connected by the CoreNet coherency fabric, sharing a 1.5MB CoreNet (L3) cache. Additionally, it has hardware support for L1 and L2 cache coherency. The design of e6500 cores also deploys many low-power techniques, including pervasive virtualization and cascading power management.

T4240RDB board provides support for small hypervisor for embedded systems based on Power Architecture technology. By using Freescale embedded hypervisor, we could add a layer of software that enables an efficient and secure partitioning of a multicore system, including the partition of a system's CPUs, memory, and I/O devices, with each partition capable of executing a different or the same guest operating

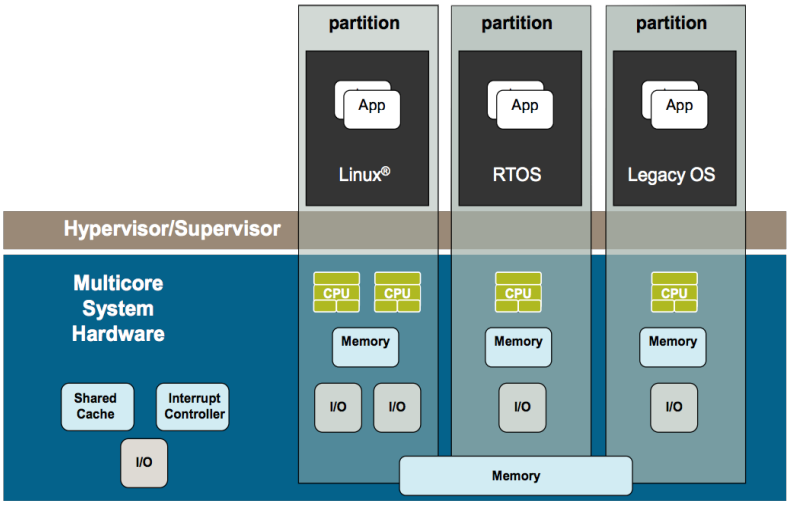


Figure 5.6: T4240RDB Hypervisor

systems. We plan to use MCAPI to exploit the hypervisor shortly. Figure 5.6 illustrates how the Freescale Hypervisor manages the embedded systems hardware and partitions on the system. Thanks to the 12 e6500 cores in T4240RDB, by deploying the Freescale Hypervisor, we could efficiently simulate Asymmetric Multiprocessing environment in many formats we would like to explore.

As a path forward we consider T4240RDB as a heterogeneous platform. We partition the T4240RDB board in two parts with each part having its hardware resource, including CPUs, memory, and I/O devices. The partitioning will help us explore how OpenMP programs can be offloaded to the partitioned system. Also, we could change the configurations to enable or disable shared memory for each partition, to evaluate the portability and performance of our runtime libraries in different scenarios. As shown on Figure 5.6, we could further explore our OpenMP

runtime based on MCA libraries when the partitions use different OSes.

#### 5.2.1.2 T4240RDB Setup

Unlike the general-purpose computer, it is a time-consuming procedure to boot up an embedded device and configure it to a certain working condition. We describe the procedures for setting up the T4240RDB board here to illustrate the efforts need to set an embedded platform.

The T4 board comes with pre-installed u-boot and the embedded Linux image on the NOR flash. Moreover, by default, it boots from the NOR flash. In the default configuration, the file system will be refreshed for every reset. We analyzed the default configuration and decided to modify the board's configuration to serve better the project.

There are some reasons we need to change the default boot mode. If adopting the default boot mode, the system will initially load the u-boot bootloader, and then read the Linux image onto the RAM memory, therefore, boot the Linux kernel in the RAM. In this scenario, any changes, we made for the board, will be completely wiped out after the system's reset. Since the file system only exists in RAM, everything stored will be lost when power is off. Due to this reason, we could not connect the board via Ethernet, but only via the serial port, making it even harder to transfer files and data to/from the board for development.

We configure the T4240RDB board to load the embedded kernel image from TFTP server while u-boot and deploy an NFS root file system to mount as shown

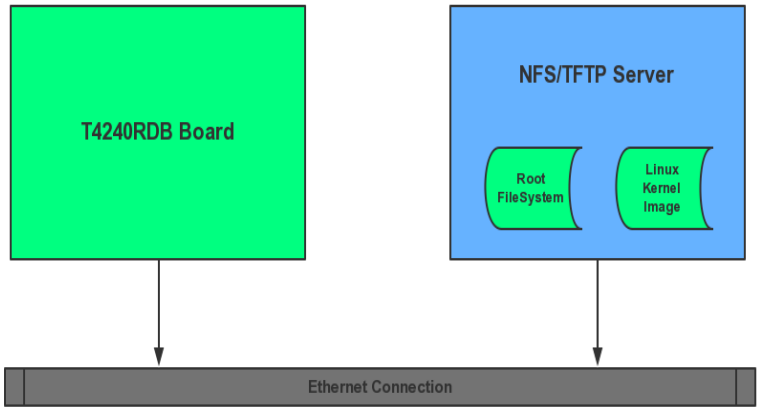


Figure 5.7: NFS Development Environment

in Figure 5.7. Trivial File Transfer Protocol (*TFTP*) is a file transfer protocol that allows a client to access a file efficiently onto a remote host [60]. In this project, the TFTP server is configured in one Linux desktop while the T4240RDB board is the client to obtain the image files in u-boot. NFS [59] is a distributed file system protocol that allows a user on a customer to access data over a network like accessing the local storage. We also configure the Linux desktop to host the NFS server to the board. By using NFS, the root file system could be customized and be saved on the remote NFS server, while overcoming the limited local hardware resources on board.

#### 5.2.1.3 Comparing T4240RDB with P4080DS

In this subsubsection, we highlight the differences in the target platform that we had used for our in 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 could be used across more than one platform, and we have substantially improved our previous toolchain to make it suitable for more than just one platform. Previously, we had used a P4080DS processor; that consisted of eight Freescale e500mc PowerPC cores. The e500mc core is compliant with PowerISA v.2.06 and includes hypervisor and visualization functionality. It supports CoreNet communications fabric for connecting cores and data path accelerators. The eight e500mc cores are attached to the CoreNet fabric directly, unlike the e6500.

Cores available on the T4240RDB platform, featuring twelve PowerPC cores that connect four e6500 cores as a cluster sharing L2 cache and then attached to the CoreNet fabric. The size of L1 cache is the same for both processors, i.e. 32KB. However, the L2 cache size is much larger for T4240 than the 128KB of unified backside L2 cache per e500mc core.

### 5.2.2 MRAPI Node Management Extension

We use MRAPI's node initialization process to create threads associated with MRAPI node IDs. MRAPI node initialization process creates new nodes related to node IDs and registers the node related information in the global MRAPI database that is shared by all the nodes in one domain. (Note: MRAPI features help abstract the hardware resources into four categories: computation entities, memory primitives, synchronization primitives, and system metadata. Such a rich feature set allows MRAPI to be used for process-level and system-level resource management covering

a broad range of use cases). Such an approach allows MRAPI to support a team of nodes where one node could be the host, and the other could be the accelerator. By extending the node initialization process with options to create threads associated with MRAPI node IDs, the procedures to deploy MRAPI on multi-thread applications, as well as the OpenMP runtime libraries, are simplified. We also modified the MRAPI reference implementation 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

As shown in Listing 5.1, the thread creation operation is accomplished for each node calling the *mrapi_thread_create* function. The function will create a worker thread for the node requested and registered the thread related information inside the global domain database for the calling node. This will be associated with the thread created and managed by the node for later use.

### 5.2.3 MRAPI Memory Management Extension

MRAPI shared memory constructs by default maps the memory allocation onto the system level shared memory, which is an Inter Process Communication (IPC) methodology. However, this is not a suitable method for OpenMP and the other thread-level parallel computation, even for embedded systems. To tackle this issue, we extend the MRAPI implementation to offer end-users more choices with memory allocation. For instance, most of the OpenMP global shared data is mapped to the process private heap instead of the system level shared memory, enabling better flexible to share among threads. Data that are mapped onto the heap could be shared by all threads incurred by the same process, which would facilitate the global data movement. Additionally, embedded devices of different types of various vendors typically support varieties of memory allocation methods. By extending the MRAPI memory model to support thread-level memory, we make it more feasible to utilize varying memory features available on different platforms.

## 5.3 Extend MTAPI onto Heterogeneous Embedded Systems

In chapter 5, we present work done 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[28] 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[30].

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

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

### 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 for a broad coverage of embedded systems. We will further explore Heterogeneous System Architectures (HSA) as a target platform for MTAPI library, by adopting HIP as the plugin for the MTAPI RTL system. This work enables the support of heterogeneous computing for MTAPI library, therefore, achieving our goal.

#### 5.3.1.1 Heterogeneous System Architectures (HSA) Background

We would firstly give a short background of HSA and its features that are of interest to us. The HSA Foundation is a not-for-profit organization of industry and academia, work to define and develop the Heterogeneous System Architecture (HSA), a set of open-sourced computer hardware specifications and software development tools. The founders of HSA Foundation include AMD, ARM Holdings, Imagination Technologies, MediaTek, Qualcomm, Samsung and Texas Instruments. Each of the founders committed to producing HSA compatible products to enrich the HSA ecosystem. Including than AMD, most of the other founders are all well known to design or product embedded System-on-Chip(SoC). As a long term goal, HSA would cover the productions from heterogeneous embedded systems, portable devices, personal computers and up to High-Performance Computers. Thus, for software that uses HSA standards, they could be executed on many different kinds of architectures without refactoring, include X86, ARM or DSPs.

#### 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 Kis 256KB while there is total 4MB L2 cache. The Kaveri APU is the first product that features HSA technology. Thus, it is the best candidate for our development and partial testing work. The technology of HSA is very new. Thus, the related support of developing HSA is quite limited so far. However, the vast potential of HSA in the future encourage us to explore and implement our system on such a cutting-edge heterogeneous platform.

#### 5.3.1.3 heterogeneous Uniform Memory Access

The HSA propose heterogeneous Uniform Memory Access (hUMA), allows CPU to pass a pointer to GPU simply; after the GPU finish the execution, CPU could read it immediately. In this case, data copy between CPU and GPU are no longer existed. The key features of hUMA is as follows:

- hUMA supports Bi-directional Coherent Memory. CPU and GPU have the same view of the entire memory space without explicit data movement to keep memory coherence.
- The GPU could access pageable memory from virtual memory that has not been shown in physical memory yet.
- Both CPU and GPU could dynamically access and allocate any location in the

system's virtual memory space.

Applications for GPUs are suffering in performance from data movement for a long time. As we can see, hUMA plays a significant role to HSA, and it has a potential to be the standard for the future heterogeneous computation.

#### 5.3.1.4 HSA Context Switch Support

The concept of context switch was not supported in the GPU industry, due to the lack of hardware support. However, HSA has proposed a software alternative to this problem. In which way, 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

As we have seen earlier, 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, as well as a simplified memory model that is connecting these devices.

In contrast to HSA, MCA APIs target to serve as the comprehensive industry standards that fully abstract the underneath system resources, enabling the power of portability over a set of embedded systems. Thus, it would be 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 device up to workstation and supercomputers, to dominate the standard for heterogeneous computation. As a promise when joining HSA Foundation, each of the founders will need to design and product HSA compatible devices. Moreover, most of these companies focus on Embedded Systems. Thus it will not be surprising that the HSA standard would be widely supported in heterogeneous multicore embedded systems shortly. Starting the release of ARM Mali GPU naming in Bifrost, the HSA feature has been supported. Due to the massive amount of shipment of the Mali devices (around 700 million devices in 2015), we will see much broader impact HSA would play for GPU computing. All the trend and evolution encourage us to extend further our OpenMP-MCA work to be able to take HIP and HSA as 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, which is a piece of the processing implemented by an action. Also, one job could be implemented by multiple actions, which could either be software-defined or hardware-defined. As shown in Figure 5.8, each MTAPI actions could take plug-ins from the external environment.

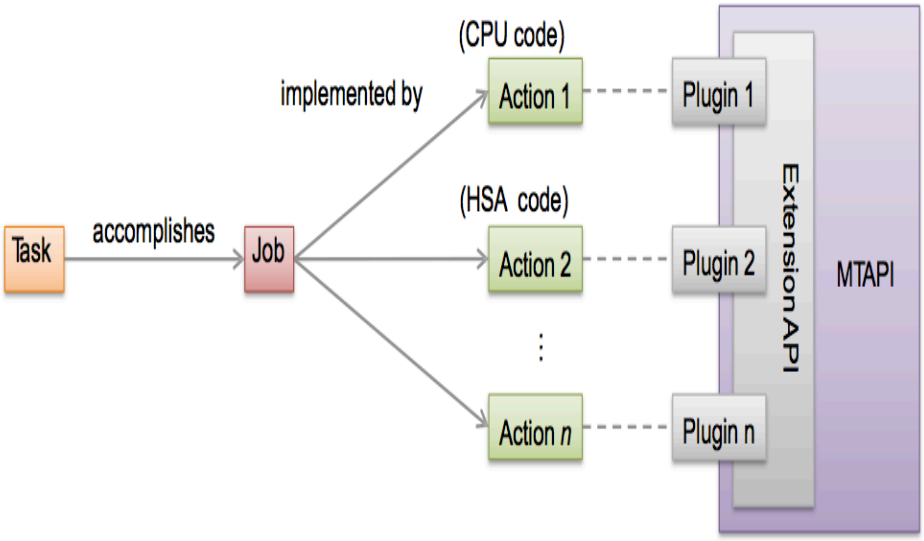


Figure 5.8: Plug-ins for MTAPI Actions

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

The detailed

### 5.3.3 Performance Evaluation Platforms

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

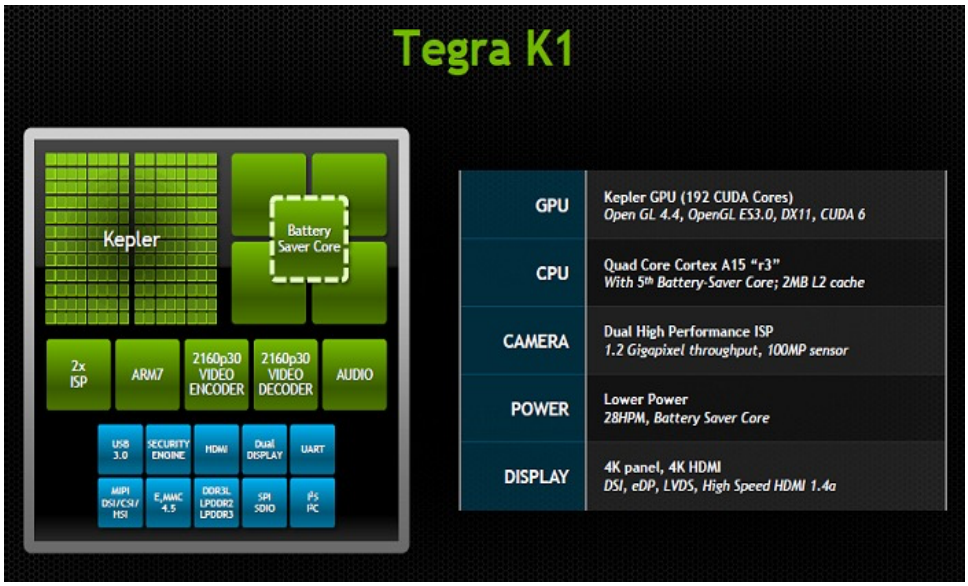


Figure 5.9: NVidia Tegra K1 Features

#### 5.3.3.1 NVidia Jetson TK1 Board

As shown in the Figure 5.9, the Jetson TK1 embedded development board features a NVidia Tegra K1 processor, which includes ARM quad-core Cortex A15 CPU and 192-core Kepler GPUs and manufactured in 28nm technology. The board also includes 2 GB of DDR3L DRAM(2 ways) and 16GB 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 devices, CPU and GPU has non-unified memory spaces. Also, there is not full hardware cache coherency between the CPU and the GPU, thus, though the CPU and the GPU only access to the same physical memory, its applications still expect the memory copy between the host CPU and

the device GPU. We have several performance evaluations in the latter sections that will illustrate the performance impact of those architecture features; and basically, we could observe longer time spent on data movement on TK1 board than on Carrizo board.

The TK1 board has been widely adopted for both research and industry, including the vast adoption of Computer Vision related domains, auto-piloted cars, and robots. Overall, the TK1 embedded board is a very powerful embedded system. Moreover, the major part of the development board is the 192 Kepler GPU cores; also, NVidia CUDA Toolkit is supported by the board. Thus, it is a great candidate for us to test the heterogeneous MTAPI.

#### 5.3.3.2 AMD Carrizo Embedded Board

As the very first device that fully complaint of HSA 1.0 specification, Carrizo was released in mid-2015. The Carrizo APU deployed on the board was manufactured in 28nm.

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

The Carrizo APU processor and its SoCs have been widely used in the embedded systems domain, including medical imaging equipment, industrial controllers, and gaming. Moreover, most of the embedded applications take full advantage of its

powerful GPU cores by using OpenCL toolchain supported on the device. With the release of HIP and HSA for the instrument, it is desired to explore the possibilities to program with such tools for better performance and easier programming interfaces.

In the dissertation, we set up the environment of the heterogeneous MTAPI and the HIP for HSA adoption on the board and achieved very competitive performance.

### 5.3.4 Performance Evaluation of Extended MTAPI Implementations

We evaluate the performance of the heterogeneous MTAPI implementation by using one set of GPU specified benchmarks and one set of applications. The first set of benchmarks are purely designed for platforms that equipped with GPU devices, and test on the features that are mostly interested to the application developers, such as the GPU streaming rates, global memory bandwidth, and PCIE data transfer rate. We will cover more in the following subsections. All those benchmarks have been ported to our platforms with HIP as the plugin.

Also, to compare the real effectiveness by adopting our solution stack and HIP for embedded systems with GPUs, we also ported a set of real applications and benchmarks utilizing GPUs as the accelerators. Those applications and benchmarks have both OpenMP versions running on the host CPUs and the HIP version ported with our heterogeneous MTAPI runtime library. Thus, in this test, we will compare 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

For this part of benchmark tests, we are mainly focusing the comparing of the full optimized powerful computation rates that could be discovered by adopting the GPU devices in the embedded systems. Therefore, the performance compares between the two heterogeneous embedded systems we discussed in the above subsections, 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 see an incredible of 512GB/sec on the AMD Fury series dGPUs equipped with the High Bandwidth Memory(HBM), comparing to 51.2GB/sec for an E5-2687 Workstation CPU processor. Thus, it is important to test on the efficiency of benchmarks for streaming applications. For the project, we ported the GPU-STREAM[25] benchmark and measured the performance on our test beds. The benchmark essentially tests the performance rate of four very 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 take minimum time on float variable computation, thus leaving the performance bottlenecks to be the global memory bandwidths of the underlying GPU

system. The performance results shown table 5.1 are measured in GB per second, and we show the results from each of the test beds on both of double precision and single precision. As a reference, the theoretical memory bandwidth of Carrizo is 25.6GB/sec, and the theoretical memory bandwidth of TK1 board is 14.928GB/sec, as mentioned in subsection 5.3.3.1 and subsection 5.3.3.2.

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, we could see the TK1 board has achieved better efficiency over Carrizo board on double precision, and slightly worse on single precision. However, due to the larger theoretical memory bandwidth on Carrizo board, we still observe big performance advantages on stream applications on Carrizo board.

**5.3.4.1.2 GPU Stride Memory Access Benchmark** As a typical GPU memory system, it is important to use coalesced memory access pattern to achieve optimal performance. 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 constructs/classes in a loop. Thus, it is important to measure the performance of the GPU devices on the stride memory access pattern.

We adopted the benchmark provided from [8], and ported to run on both TK1 and Carrizo embedded boards.

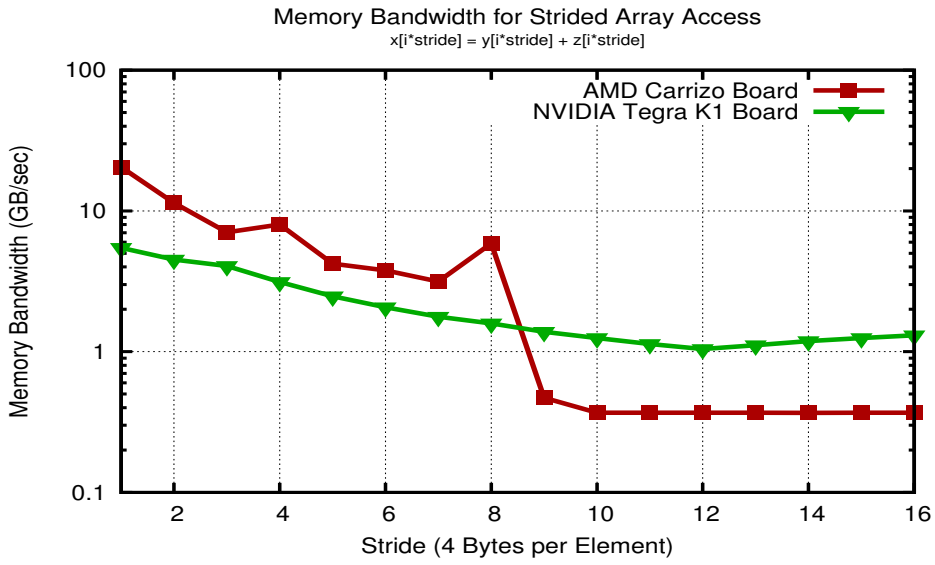


Figure 5.10: Stride Memory Access Benchmark Results

As shown in 5.10, we see the performance on Carrizo board has a certain advantage when the size of stride is less than 8 bytes. However, when the size of stride increases to more than 8 bytes, the performance on Carrizo decreased dramatically. The behavior relates to the wavefront size of AMD GCN architecture and the optimization on the driver-level support.

For GPU architecture, *warp* for NVidia GPU and *wavefront* for AMD GPU are the basic unit of hardware scheduling. The warp for NVidia GPUs consists 32 threads and the wavefront for AMD GCN GPUs are aligned with 64 threads. Each of the warp/wavefront for GPU applications process a single instruction over all of the threads in it at the same time. Thus, when the stride size is larger than 64bits(8bytes), there will not be any data reuse inside the wavefront for each read of data from the global memory. It could explain part of the performance decrease

when the stride size larger than 8 bytes for Carrizo board.

In comparison, the performance on TK1 board is more stable with different stride pattern. The good performance should be related to the optimizations on the NVidia tool chains and drivers for the stride memory and texture memory access. However, because of the CUDA toolchain is proprietary, we have no way to get a deeper investigation into any potential optimizations.

Here is the takeaway from this benchmark evaluation: for Carrizo board, we should limit the size of stride memory access less than 8 bytes or 64 bits, to achieve good performance. Moreover, there is a good room to be improved for AMD drivers for larger stride memory access for the future vendor support.

**5.3.4.1.3 Mixed Benchmark for GPUs** In the previous GPU specific benchmarks, we have evaluated the data streaming and the stride memory access rates, to extract several characteristics of the two test beds. Besides, we ported the mixbench [40] to measure the performance of GPUs on a mixed operational intensive kernel. The benchmark contains mixed operations of multiply and add, and configured for the data type of float, double and integer.

As the increase of iterations for each set of tests, more load of data would be fed into the computation kernels. As shown in Figure 5.11, the Carrizo board has a significant lead on the overall performance for each of the float, double, and integer data types. For float type benchmark 5.11a, we see a rather stable performance for iterations smaller than 32; and both on Carrizo and TK1, the performance starts to decrease at a similar rate for iterations larger than 64. On double precision 5.11b and

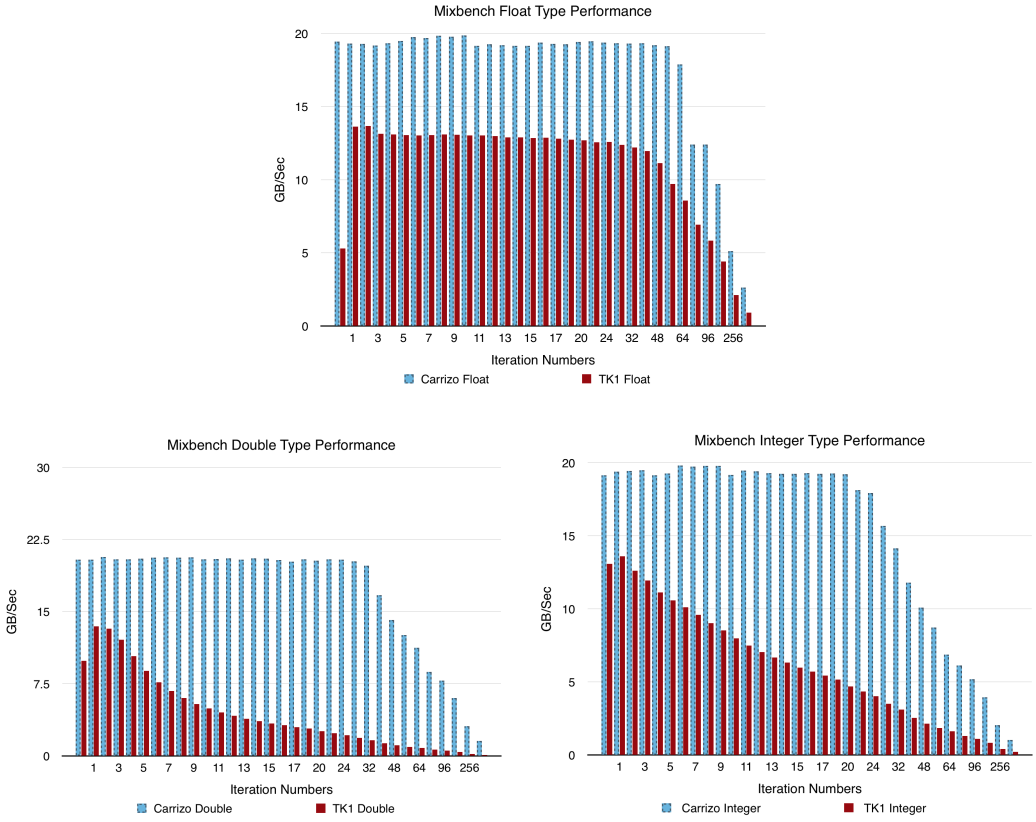


Figure 5.11: Evaluating Mixbench Benchmark for Carrizo and TK1 Boards

integer 5.11c tests, the performance on Carrizo board are very stable on iterations smaller than 32, and start to decrease gradually for iterations after so. However, for TK1 board, we see the performance decrease starts as early as the second iteration, and the decrease rate for double precision is similar to the normal distribution, and the rate for integer type is rather linear. The performance advantage on Carrizo board would be mostly due to the efficient HSA compliant architecture, especially for its hUMA memory for faster data coherency and copy rates, as well as the hardware kernel dispatch. All those architecture features could be minimum the overheads for kernel executing on the GPU cores, thus better performance.

With all the three GPU specific benchmarks we have ported to our solution stack, we could conclude the main features from the two test embedded boards. TK1 board from NVidia is very powerful, and we could see the performance data collected from this platform are more stable and with more mature CUDA tool chains. However, the nature of its proprietary software support results that we have limited room to explore on the performance data we have observed from the benchmarks; thus, we could dive deeper into the underlying system.

We also observe that, by adopting the HSA architecture, we could expect a good performance on the Carrizo board. However, due to the HIP, the plugin we have taken, are relatively new and not as mature as NVidia CUDA toolchains, we see much room to be improved for the compiler and driver support for the future. We have to emphasize, the HIP and ROCm stack for it are all open sourced; unlike CUDA toolchains, we have the flexibility to explore deep throughout the stack and tune for performance as need. As we would like to convey in the dissertation, 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 from a set of real applications and benchmarks. For this round of test, we mainly focus on the performance gain by dispatching tasks or workloads onto the GPU accelerators. Thus, the comparison would mainly target the host side OpenMP performance and the absolute performance across platforms.

**5.3.4.2.1 LU Decomposition Performance** The LU Decomposition application is one implementation of an algorithm that calculates the solutions of a set of linear equations for the input matrix. The application is a challenge for its relatively big row and column dependencies through the input matrix. The application is modified for using HIP and our heterogeneous MTAPI; we will also discuss in the latter chapters for its performance combined with OpenMP task constructs. We analysis the details of the application performance by comparing the time consumed for data movement time between the host the device, kernel execution time and the overall CPU time between the Carrizo board and the TK1 board. Moreover, we also compare the over all performance between the parallel host-side and execution on the GPU accelerator.

As shown in the Figure 5.12, the overall execution time on Carrizo board, is much smaller than that on the Tegra K1 board. Furthermore, if we look into the details

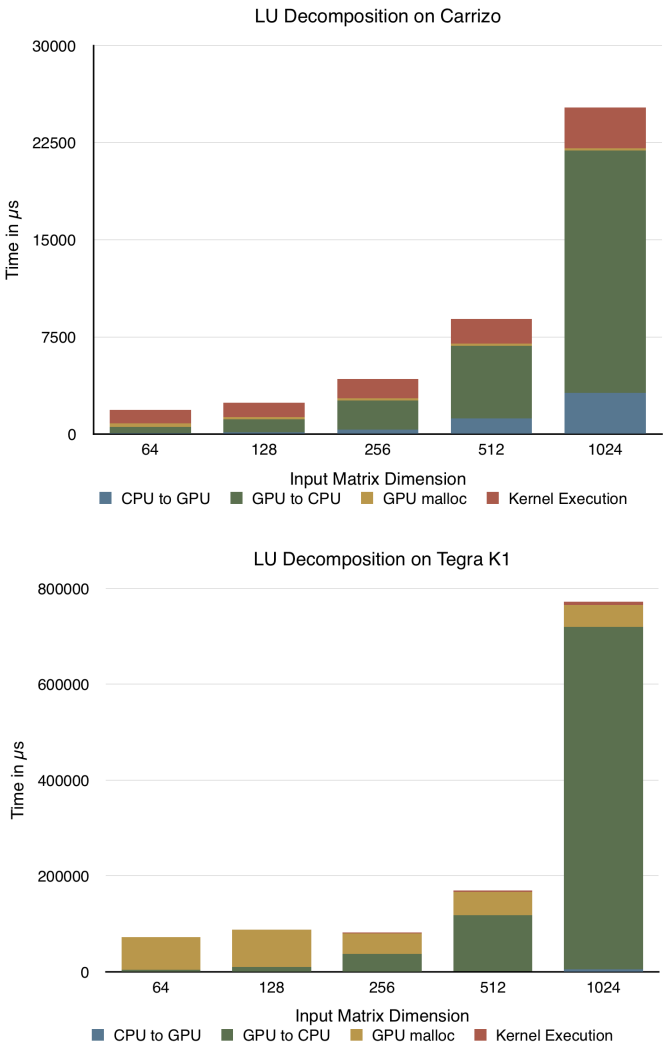


Figure 5.12: LU Decomposition on Carrizo and Tegra K1 Boards

of the time taken on each step of the execution from the Figure 5.12, the difference mainly dues to the data allocation and movement, but not on the kernel execution time.

The time spent on kernel execution for both of the embedded boards are quite similar. For analysis, the fast data transfer on Carrizo board is mainly due to the hUMA memory architecture we have discussed, as part of the HSA architecture features. In 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 Carrizo board is only means to pass on the pointers between the host CPU and the target GPUs.

However, on the NVidia TK1 board, the CPU and GPU units are logically two elements in the system that have separate memory space in the same physical memory. Thus, the data transfer are enforced before and after the kernel execution, and those steps take the primary partition of the executed. This test is a good example to show the advantage to support bi-directional memory coherence for a typical CPU and GPU-equipped computing system.

To explore the benefits by extending our MTAPI onto GPUs, we further explore the potential speed gains we could achieve comparing to execute the application completely on the host side. We run the OpenMP version of the LU Decomposition application on the host CPU side with four threads, on both of the Carrizo board and the TK1 board. The result has been illustrated on the Figure 5.13. We could see that, due to the requirement of explicitly data movement on the TK1 board, we could not observe any performance gains when comparing the performance on

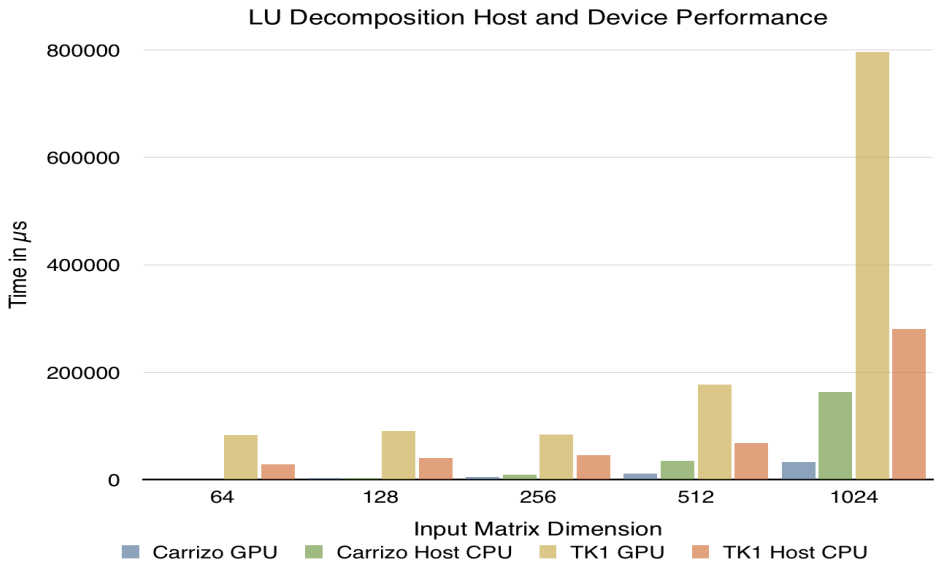


Figure 5.13: LU Decomposition Performance CPU vs GPU

the TK1 host side to the GPU side. However, we did see a good level performance gain on the Carrizo board. As the size of the input matrix increase, we see a bigger performance advantage on the Carrizo GPU side. The trend we observed is mainly because the massively parallel computation capacity provided by the GPU.

However, the data movements for GPU systems are very costly and requires very careful optimization and performance tuning. Moreover, the HSA compliant features provided by Carrizo could minimize this concern. We have achieved better overall performance on the Carrizo board than the TK1 board, which would highly ascribe to the HSA architecture.

With the applications we have explored in this section, we could conclude that, by enabling HIP as the plugin, we could extend our MTAPI RTL to support the

heterogeneous embedded systems featuring GPUs. Furthermore, we could achieve much better performance on the HSA compliant architectures, due to the minimum requirement of data movement on such systems.

# Chapter 6

# Map OpenMP onto MCA APIs

To complete the solution stack we proposed for portable programming onto a various heterogeneous embedded systems, we explored the MCA APIs in the previous chapter. In this chapter, we introduce our work that maps OpenMP, the higher-level programming models onto the MCA APIs we have implemented, thus, to achieve the goal to provide 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 library. We had built an OpenMP runtime library called *libEOMP*[70] where we had mapped essential resource management functionality of the MCA libraries to an OpenMP runtime library implemented in OpenUH compiler[43]. OpenUH translates OpenMP directives and function calls into the multithreaded code for use with a custom runtime library (RTL). GCC's latest release provides support for OpenMP 4.0 for C/C++ compilers and FORTRAN, which is encouraging for multicore embedded programmers to go further with 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 be converted into a parallel code. After translating the OpenMP constructs to C-like IR from original pragmas, 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, managing threads pool, and the synchronization among threads. Besides, an efficient OpenMP runtime can also offer productive scheduling, data locality, and workload balancing techniques.

Most of the cases in general purpose computation, the systems offer fully featured OSes and a unique multi-threading library available to be effectively utilized by the OpenMP runtime library. Unfortunately, most of the embedded systems do not necessarily 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.

### 6.1.1 Memory Mapping

In the OpenMP program runtime, a set of global data structures needs to be maintained. For example, each team of nodes would need to keep a block of work share, to be assigned to each node later for computation. In this project, we extend the MRAPI shared memory constructs to be able to allocate thread-level shared data, as shown 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 RTL has been mapped to MRAPI Mutexes, preventing critical data races and managing accesses to the shared data. Specifically, we use the *mrapi_mutex_create* function to create the Mutex object as the initialization step, and map the Mutex lock and unlock functions to MRAPI

Mutex routines as well.

```
/* 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 routine we used to enhance the native libGOMP RTL. MRAPI Mutex routine will map the lock operation for the target system, thus making this low-level operations portable according to a various set of systems supporting MCA API.

### 6.1.3 Metadata Information

MRAPI metadata constructs have also been utilized within the OpenMP runtime library. We mainly used the MRAPI metadata trees to retrieve the available number of processors online for node/thread management, to better serve the MRAPI nodes

and threads allocation and management accordingly.

### 6.1.4 Evaluating GNU OpenMP with MRAPI

In this subsection, we have conducted experiments to measure the performance of the extended GNU OpenMP runtime library with MCA API, discussion on the overhead and performance results follows.

Table 6.1: Relative overhead of OpenMP-MCA RTL 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 |

To measure possible overheads caused by enhancing GNU OpenMP runtime library to MCA API, we use EPCC[17] to evaluate the OpenMP-MCA RTL. EPCC is a set of programs that measure the overhead of each of the OpenMP directives and evaluates different OpenMP runtime library implementations. Table 6.1 lists the results for OpenMP-MCA RTL compared with the native GNU OpenMP runtime. In the table, we normalize the overhead numbers of OpenMP-MCA RTL divided by the original overheads number, to provide a relative performance number in the table; the smaller number indicating fewer overheads. The table illustrates that OpenMP-MCA RTL does not incur major overhead, and it performs even better for some constructs.

With different thread pool sizes, the PARALLEL construct performs better than libGOMP; while the overheads are slightly higher than libGOMP for the *for* construct. Thus, the combined *Parallel for* construct has similar overhead performance. The rest of the constructs are also comparable to libGOMP with different size of threads pool. The performance comparison illustrates that we have achieved competitive performance for each of the OpenMP constructs on T4240RDB board while OpenMP-MCA RTL still has more room to be optimized with a larger thread pool.

Next, we wanted to ensure the correctness of our implementation. For this step, we used our OpenMP validation suite [69] to identify if the enhancements made to the runtime did not cause code to fail. The results helped determine some bugs, and we fixed them such as tracing potential issues with a non-functional synchronization primitive in OpenMP-MCA RTL that caused an OpenMP critical construct to fail.

We then evaluated our OpenMP-MCA RTL implementation using NAS OpenMP benchmarks [37], the results are illustrate in Figure 6.1. The execution time is in seconds, and the performance comparison is between the OpenMP-MCA RTL runtime library and the proprietary GNU OpenMP runtime library. The NAS OpenMP benchmarks have several different data sets available to choose from. Typically, size S and W, which are the smallest data sets available, could be used to validate the correctness of the compiler being tested and the runtime library. The larger data sets could be used to measure the real 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 runtime for a single thread is measured in several seconds.

We illustrate the performance of a single thread to 24 threads, which is the maximum amount of threads available on the T4 board. Besides the performance comparison, we also measure the speedup rate of both runtime libraries within the same graph.

As shown in Figure 6.1, the performance of OpenMP-MCA RTL is very comparable to the proprietary GNU OpenMP runtime library. In CG, EP, and IS, the OpenMP-MCA RTL runtime library shows better performance compared to proprietary libGOMP for a specified number of threads. On the speedup rate, most of the benchmarks perform well. Both the OpenMP runtime libraries are close to the ideal speedup rate for benchmark EP. The rest of the benchmarks could achieve speedup around 15 times using 24 threads.

The performance and speed up rates shown in Figure 6.1 could prove that, by enhancing libGOMP with MCA libraries we did not incur any significant overheads, but provided a portable software toolchain that uses a high-level programming model, OpenMP to create a multithreaded application translated to industry-based standard MCA to target a multicore embedded platform.

## 6.2 Mapping OpenMP Task Parallelism to MTAPI

In this section, we discuss our proposed light-weighted, portable OpenMP runtime library (RTL) that maps its parallelism task functionalities onto MTAPI. Task parallelism is essential for embedded products. The automotive application is a good example. The state-of-the-art automobile MCU need to handle different signals and

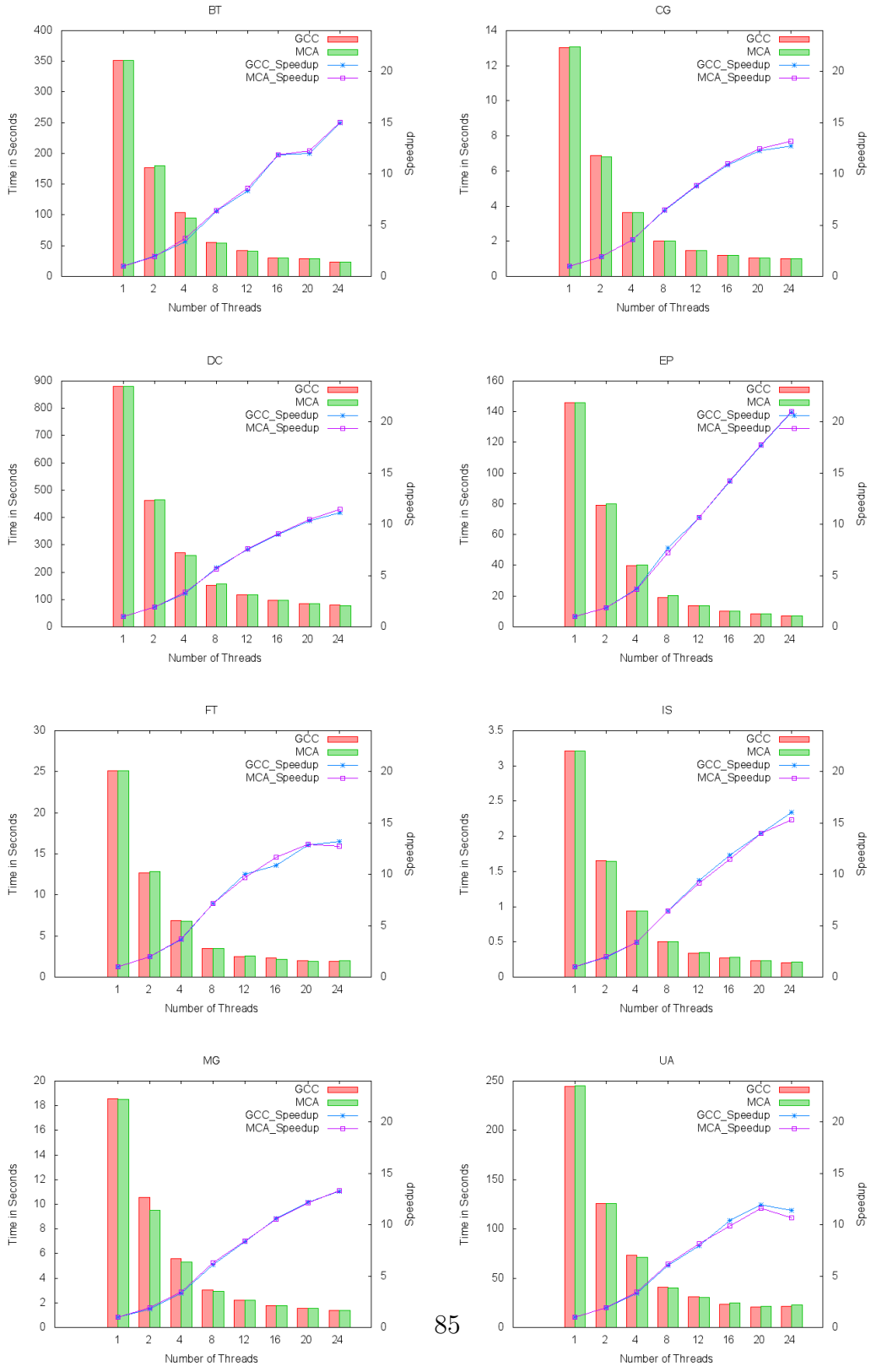




Figure 6.1: Evaluating OpenMP-MCA RTL runtime library using NAS benchmarks

executions from many various aspects, such as electronic controls units for valves, fuel injection, sensors for various aspects of information and driver assistance. Each of these operations could be classified as a task, thus has further potential to be executed in parallel with the help of an adequate programming model, such as OpenMP task-based model. Moreover, the vast potential of self-driven cars exacerbates the needs for task parallelism. There are many other examples available in the embedded domain that would require the help of task parallelism, including robotics and aircraft. However, due to the impediments we discussed in the above chapters, to map OpenMP task constructs onto embedded devices is difficult, to build a portable implementation of a set of devices with different architectures or instruction sets would be even harder. Thus, in this section, we used the industry-standard task management API (MTAPI) as the mapping layer of OpenMP task constructs and achieved competitive performance results.

### 6.2.1 Overall framework

In subsection 6.2, we introduce our previous work that maps OpenMP task parallelism support onto MTAPI to map it to embedded systems by building a portable and light weighted runtime library (RTL). We have mapped OpenMP on embedded systems by mapping it on MCA APIs. Besides, as we discussed in the above sections, MTAPI can take plug-ins to implement actions that could be executed on devices on different architectures. With the heterogeneous MTAPI library we discussed in section 5.3, we will be able to extend further our solution stack onto heterogeneous platforms.

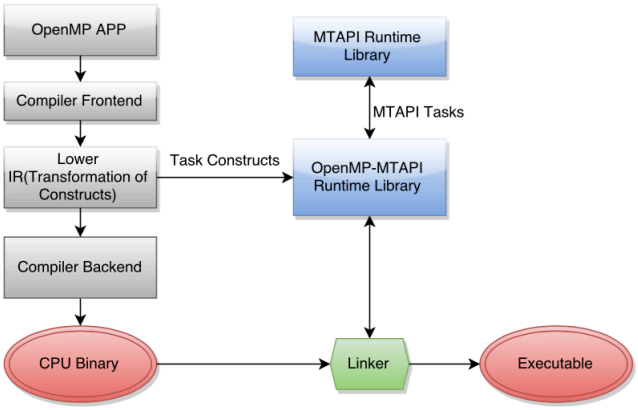


Figure 6.2: OpenMP-MTAPI Solution Diagram

Our proposed solution is to map the OpenMP constructs to MTAPI APIs to provide an elegant OpenMP programming interfaces and a comprehensive software stack for embedded systems. This work requires a broad learning curve of the OpenMP compilers and their runtime translations, as well as the environment configurations. Figure 6.2 shows our framework. An OpenMP compiler front-end translates the OpenMP constructs into OpenMP-MTAPI runtime function calls, therefore, the lower IR contains the transformation of the OpenMP application. The executable file is formed by linking OpenMP-MTAPI RTL and MTAPI RTL by the system linker. During the runtime of the application, the OpenMP-MTAPI RTL takes the OpenMP task function pointer and the task data pointer to create an explicit task. The OpenMP-MTAPI RTL then translates the explicit task onto MTAPI task, moves the control of execution onto the MTAPI RTL. After a parallel execution, the MTAPI RTL sends the results back to the OpenMP-MTAPI RTL, thus accomplishing the OpenMP task.

### 6.2.2 Implementation

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

#### 6.2.2.1 Parallel Construct

In the conventional OpenMP RTL, when the master thread encounters an OpenMP parallel region, a set of worker threads will be created and be associated with a team. Thus, this is the "fork" of OpenMP's "fork-join" execution model. After the creation of worker threads, the OpenMP runtime will set them into the state of *wait* that could later be used to wake the worker threads up to a working state as and when needed. This way the overheads of the OpenMP construct would be reduced since threads do not need to be created multiple times.

In our enhanced OpenMP - MTAPI RTL, we intend not to handle threads directly; this is to ensure implementation portability. Instead, we use MTAPI solely to control the thread management and workload scheduling, to ensure the RTL's portability across different architectures and OSes. In our RTL, we initialize the default MTAPI node and create the default MTAPI action with the associated job handle when codes encounter the parallel region. Therefore, the program can start MTAPI tasks without further initialization later.

#### 6.2.2.2 Task and Taskwait Constructs

OpenMP compilers translate each of the explicit tasks to an RTL function call with several parameters, including function pointer, data frame, arguments, and dependencies. Inside the RTL function, the tasks will be created and put into the task queue for further execution. For *taskwait* construct, a typical OpenMP RTL will specify the descendant tasks of the current task, and wait for their completion before moving to the next step.

We directly map OpenMP explicit tasks to MTAPI tasks in OpenMP - MTAPI RTL, 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, and storing the returned task handle for further reference of the tasks created. We also map the optional OpenMP group ID to the MTAPI tasks. In the context of MTAPI, *task wait* needs to utilize the task handle obtained from the creation of tasks and then waits for the completion of the corresponding task. Thus for the mapping of OpenMP tasks, we assign the task wait for the tasks created in the context and allow the completion of their child tasks, to meet the requirement of OpenMP taskwait construct.

#### 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 descendant tasks belong to the same

taskgroup. At the end of the taskgroup region, there is a synchronization point, thus forcing all tasks in this taskgroup to wait for completion before continuing execution.

MTAPI specification provides a set of routines for taskgroup management. In the process of creating MTAPI tasks, programmers have options to whether or not to associate the tasks to a taskgroup. In our OpenMP runtime implementation, when encountering OpenMP taskgroup region, we create an MTAPI task group at the runtime with the default group ID and collect the returned group handle reference. Inside the group region while creating tasks, we specify which MTAPI taskgroup the tasks belonged to, by using the option to specify the taskgroup handle in the *mtapi_task_create* function. We also translate the OpenMP runtime call that refers to the We also translate the OpenMP runtime call that referring to the end of taskgroup onto the *mtapi_group_wait_all*. This is performed with the taskgroup handle previously obtained when creating the task group, to wait for all the tasks and their child tasks in the group.

### 6.2.3 Performance Evaluation

In this subsection, we evaluate the implementation of our OpenMP - MTAPI RTL. This is to demonstrate our implementation could still achieve comparable performance while ensuring portability. We use the OpenMP Task micro-benchmarks[41] to demonstrate the overheads incurred by the OpenMP task construct. We discuss the performance analysis by evaluating real applications from OpenMP task benchmarks from Barcelona called the OpenMP Task Suite (BOTS)[27]. The experimental

platform consists of two E5520 CPUs, with 16 threads available for execution, thus could provide sufficient room to explore the performance of OpenMP-MTAPI RTL over different numbers of threads. As mentioned earlier, we use the EMBB MTAPI implementation for our prototype OpenMP-MTAPI RTL.

#### 6.2.3.1 OpenMP Task Micro-benchmark Measurement

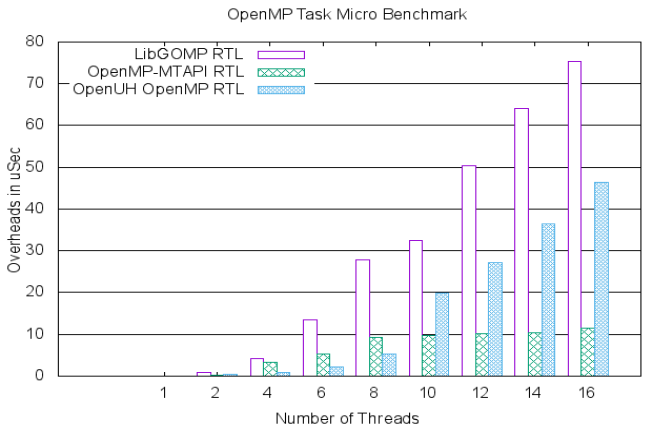


Figure 6.3: OpenMP Task Overheads Measurement

To measure the performance of our translation, first up we find out if the addition of an MTAPI layer resulted in any overhead. We use the OpenMP task micro-benchmark suite and compare the overheads of our OpenMP - MTAPI RTL against GCC OpenMP runtime library and OpenUH OpenMP runtime library[9]. The overhead in this scenario is the difference between parallel execution time and the sequential execution time over an identical section of codes. Thus, during testing, the micro-benchmark conducts a large number of iterations, for both task-parallel executions and subsequent executions, therefore calculating the average differences

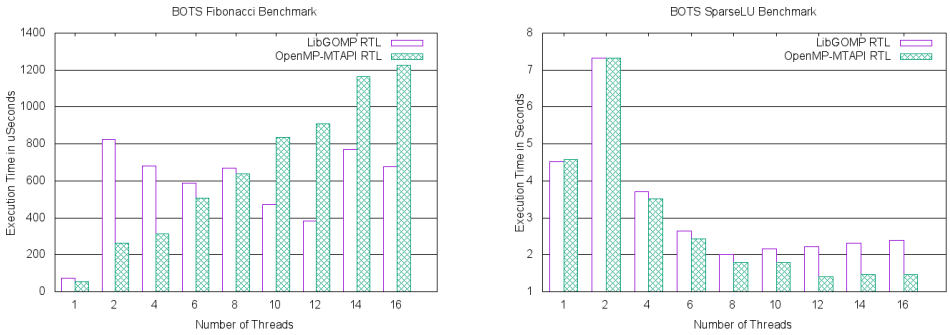


Figure 6.4: Evaluating OpenMP-MTAPI RTL with BOTS benchmarks

and the overhead measurements. Figure 6.3 illustrates the comparison of the overheads. In the graph, the Y-axis indicates the actual overheads of the OpenMP task constructs in microseconds, and over threads ranging from 1 to 16. The OpenMP-MTAPI RTL performs similarly to the GCC OpenMP RTL for threads less than 4. However, we notice that the OpenMP-MTAPI RTL has significant advantages on a larger number of threads. The performance of OpenUH OpenMP RTL has evidently fewer overheads for up to eight threads; however, when the number of threads increases to a larger amount, the OpenMP-MTAPI RTL achieves better performance.

The main reason for the proper performance on the OpenMP task overheads is due to MTAPI, particularly the implementation brought by the EMBB from Siemens. Minimum overheads of tasks creation and scheduling over a larger amount of threads is desired when MTAPI is deployed on multicore embedded devices.

In this subsection, we measure the performance of the OpenMP-MTAPI RTL using the BOTS suite. We choose the Fibonacci and SparseLU for experimental analysis and 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 in tasks, but heavy dependency and synchronization among the tasks. As shown in Figure6.4a, the OpenMP-MTAPI RTL performs better than GCC OpenMP RTL with less than eight threads while GCC OpenMP RTL has a distinct advantage with a larger number of threads. This shows our prototype implementation could be further improved, especially the synchronization strategies when using many threads.

#### 6.2.3.3 SparseLU Benchmark

The SparseLU benchmark computes an LU matrix factorization over sparse matrices. The matrix shows a lot of workload imbalance; using task parallelism and dynamic scheduling could lead to a good performance.

## 6.3 OpenMP-MTAPI for Heterogeneous Embedded Systems

To further extend our solution stack onto heterogeneous embedded systems, we design and develop the heterogeneous MTAPI runtime library in Section 5.3. Moreover, to measure the performance of the extended heterogeneous MTAPI RTL, we have ported two sets of benchmarks and applications using the extended MTAPI RTL and achieve relatively good performance on our targeted HSA compliant Carrizo

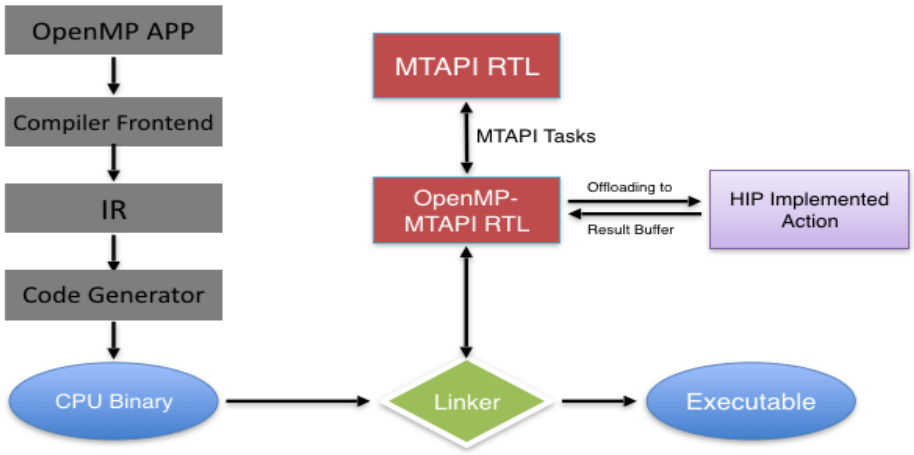


Figure 6.5: OpenMP to MTAPI Mapping with HIP Plug-in

embedded board. In this section, we introduce our work to use the OpenMP task constructs with our heterogeneous MTAPI runtime library, to enable 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 RTL flowchart, which takes HIP implemented action on the MTAPI, thus be able to offload computations onto the targeting devices. In our implementation, the communication and offloading tasks to HIP and underlying HSA platforms will be completely handled by MTAPI library. The role the OpenMP-MTAPI RTL plays is to be the intermediate layer that dispatches from OpenMP tasks construct onto MTAPI task queue and specifies the dependencies between tasks.

Thus, when an explicit task is created in the OpenMP task context, the runtime library will gather the related information of the task and then push it into the task queue handled by the MTAPI runtime. Based on the information provided for the task, MTAPI RTL will schedule the task in a proper way and marshal the tasks to execute the action that whether from the host or the offloading device, in this case, the HSA devices and Nvidia GPUs that enabled by HIP. The development platform would be the same as we discussed in5.3.1.2, and we use the same test beds we have adopted in subsection5.3.3, the Carrizo board that features of HSA compliant APUs, and Tegra K1 board from Nvidia, which features ARM CPUs and 192 Kepler GPUs.

### 6.3.2 Performance Measurement for Heterogeneous OpenMP-MTAPI

In this subsection, we measure the performance and efficiency of benchmarks and applications that use our solution stack for heterogeneous platforms. The underlying heterogeneous embedded platforms are the Carrizo board featuring full HSA compliant APUs and Tegra K1 board that consists of four core ARM CPUs and 192 Kepler GPU cores. We analysis each of the benchmarks and applications by comparing the performance on CPU only and with GPU for accelerating. The time spent on data transfer and kernel execution are also considered in the tests.

To evaluate the performance of our extended solution stack with heterogeneous MTAPI implementation, we port several benchmarks and applications from Rodinia[20] and the Scalable Heterogeneous Computing (SHOC)[24] benchmark suite, which are

Table 6.2: RTM8 Application

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

the well-adopted heterogeneous benchmark suites for industry and academia.

The main purpose of the test is to measure the impact and change that would potentially introduce by extending MTAPI onto heterogeneous platforms and the potential of any overhead by adding OpenMP task construct on top of it.

#### 6.3.2.1 Grid Computation Evaluation

RTM8 is modified from an oil and gas industry application and ported to our solution stack. We use this application to evaluate the potential of speedups for such structured grid applications, and since the structure of the application fits into the massive parallelization nature of GPU devices, we expect good performance for this application. The original application is programmed in FORTRAN, we use the single thread FORTRAN code as the base of the performance comparison.

**6.3.2.1.1 RTM Application** As shown in Table 6.2, we could observe big performance gain by porting the application to GPUs using our solution stack. The speed up of the performance on GPU is 229.26 times faster than that on single thread host CPU. Since this application involves minimum data transfer between

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 |

host and device, we could also obtain good performance on the TK1 board, about 82% of what we could get from the Carrizo board. This application could illustrate that, for a certain application that requires a large number of structured grid operations, we could potentially benefit a lot by porting the applications onto GPUs.

**6.3.2.1.2 Hotspot Benchmark** HotSpot is a widely used algorithm that estimates processor temperature based on an architectural floorplan and simulated power measurements. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. We modified the HOTSPOT benchmark to work with heterogeneous MTAPI RTL and be marshaled 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, while using CZ to account for the performance on Carrizo board. The time for each step is measured in s, and the Speedup is calculated by dividing time of OpenMP on the host to the time of benchmark offloaded to GPU.

By analyzing the performance table, we could observe that the HOTSPOT benchmark could achieve much better performance on the Carrizo board, with an average speed up by the factor of four. While the benchmark runs very poor on the Tegra K1

board. If we look deeper, we could see that the data transportation and device data allocation takes the majority of the execution time. Thus, though the kernel execution time on both Carrizo board and Tegra board are very similar, the difference on overall performance are huge. The reason behind is that, on the Carrizo board, the hUMA memory architecture and HSA architecture are enabled. Thus there are theoretically much smaller overheads to move data from the host to and back from the device, as well as data allocation. With the benchmark, we could conclude that by adopting proper software tools and underlying heterogeneous embedded system, better performance could be achieved by offloading part of the execution onto accelerators. Our solution stack could help programmers to explore parallelism over devices across vendors, thus, provide better performance and portability.

# Chapter 7

# Conclusion

## 7.1 Conclusion

This dissertation focuses 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, resource management, and task management APIs to move data between cores, query resources and create and schedule tasks on the 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 complaint to MCA APIs can benefit from OpenMP. As we know, embedded systems is typically a heterogeneous platform consisting of either ARM + GPU, or ARM + DSP, or CPU + FPGA or CPU + GPU; it is critical to extend the software stack to support such types of systems. These provide cores of varying functionality

and features and without an efficient software stack, it is a challenge to tap the potential of these cores.

To address these many problems, we are proposing a solution stack that will map OpenMP to heterogeneous embedded systems by adding an HSA plug-in for the MTAPI RTL. This will allow us to offload computations to HSA compatible devices. Since we are utilizing cores of different types, to obtain better performance, we propose to develop a smart scheduling strategy that will enable appropriate task distribution across the platform. This scheduler will analyze scheduling decisions based on several characteristics of the tasks and the underlying computing 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 platforms.
- Evaluate the proposed solution stack on a various heterogeneous embedded systems and show the effectiveness and portability.

## 7.2 Future Work

For my dissertation project, one future work includes the exploration of more adoptable targeting heterogeneous embedded devices, including bare metal devices, DSP

processors and FPGA devices. Those sets of devices are very common in embedded devices, as part of embedded systems, and play crucial roles for a various of applications scenarios. However, the programming models for those devices are still lacking. We believe our solution stack, in this context, would be helpful for abstracting the lower level systems details and coordinate the execution over the underlying heterogeneous computation units.

Besides, with the release of OpenMP 4.5, the support of computation offloading on devices are better supported by the OpenMP specification. Thus, it is desirable to explore the OpenMP *target* and *target data* construct in combination of the MCA APIs onto heterogeneous systems. We believe OpenMP would be the proper standard high-level programming model for future heterogeneous computing on embedded platforms.

In addition, more combination of programming tools is necessary for the solution stack to involve more efforts for a wider adoption. We see the trend of adopting more GPU elements in the current embedded systems design, especially for the industry of auto-driving car, medical imaging, and Virtual Reality. The effort of introducing HIP as the plugin in the project is our first step to easy 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] Multicore Association Website. http://www.multicore-association.org.

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

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

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

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

[8] 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).

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

[10] 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.

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

[12] 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.

[13] D. A. Augusto and H. J. Barbosa. Accelerated parallel genetic programming tree evaluation with opencl. *Journal of Parallel and Distributed Computing*, 73(1):86–100, 2013.

[14] D. F. Bacon, R. Rabbah, and S. Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] A. Kamppi, L. Matilainen, J. Maatta, E. Salminen, T. D. Hamalainen, and M. Hannikainen. Kactus2: Environment for embedded product development using ip-xact and mcapi. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 262–265. IEEE, 2011.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

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

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

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

[57] 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.

[58] J. Sérot, F. Berry, and S. Ahmed. Caph: A language for implementing stream-processing applications on fpgas. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer, 2013.

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

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

[61] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[62] 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.

[63] T. Suganuma, R. B. Krishnamurthy, M. Ohara, and T. Nakatani. Scaling analytics applications with opencl for loosely coupled heterogeneous clusters. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 35. ACM, 2013.

[64] 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.

[65] 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.

[66] 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.

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

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

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.