Exploring Task Parallelism for Heterogeneous Systems Using Multicore Task Management API

Suyang Zhu[1], Sunita Chandrasekaran[2], Peng Sun[1], Barbara Chapman[1], Marcus Winter[3], and Tobias Schuele[4]

[1] Dept. of Computer Science, University of Houston
zsuyang@uh.edu, psun5@cs.uh.edu, chapman@cs.uh.edu
[2] Dept. of Computer and Information Sciences, University of Delaware
schandra@udel.edu
[3] Hypnolords Gbr
marcus.winter@hypnolords.com
[4] Siemens Corporate Technology
tobias.schuele@siemens.com

Abstract. Current trends in multicore platform design indicate that heterogeneous systems are here to stay. Such systems include processors with specialized accelerators supporting different instruction sets and different types of memory spaces among several other features. Unfortunately, these features increase the effort for programming and porting applications to different target platforms. To solve this problem, effective programming strategies that can exploit the rich feature set of such heterogeneous multicore architectures are required without leading to an increased learning curve for applying these strategies.
In this paper, we explore the Multicore Task Management API (MTAPI), a standard for task-based parallel programming in embedded systems. MTAPI provides interfaces for decomposing a problem into a set of tasks while abstracting from the heterogeneous nature of the target platforms. This way, it allows to efficiently distribute work in systems consisting of different cores running at different speed. For evaluation purposes, we use an NVIDIA Jetson TK1 board (ARM + GPU) as our test bed. As applications, we employ codes from benchmark suites such as *Rodinia* and *BOTS*.

Keywords: Multicore Systems, Runtime, Heterogeneity, Accelerators, MTAPI

1 Introduction

Embedded multicore systems are widely used in areas such as networking, automobiles, and robotics. Some of these systems even compete with HPC platforms [19] and are promising to deliver high GFLOPs/Watt. Since parallelism has become a major driving force in computing, microprocessor vendors concentrate on integrating accelerators together with the CPU on the same platform. The current trend of such platforms is that they are heterogeneous in nature, i.e.,

their instruction sets and memory spaces are usually different [13]. For example, Qualcomm's heterogeneous processor Snapdragon 810 integrates an ARM Cortex CPU and an Adreno 430 GPU on the same chip. Such an integration produces hardware platforms that satisfy the requirements regarding performance, flexibility, and energy consumption of embedded systems. Unfortunately, the available runtime systems and tools to port applications to other platforms are often still immature and typically fine-tuned for specific product families. As a result, maintaining a single code base across multiple target platforms is hardly feasible, which is a major concern in the industry.

Programming models designed for high performance computing (HPC) platforms are not necessarily the best for handling embedded multicore systems, especially when these systems have limited resources or are subject to real-time constraints. Among these models, OpenMP [7] has been recently extended to support accelerators. In its current state, however, it still lacks support for the requirements in embedded systems. An alternative is OpenCL [18], which is known for its portability but also involves a steep learning curve making it a challenge to adopt applications to new hardware.

The Multicore Association (MCA)[5], formed by a group of companies from different domains, aims to address these challenges by providing a set of open specifications for programming embedded multicore platforms. MCA has released APIs for sharing resources among different types of cores via the Multicore Resource Management API (MRAPI), inter-core communication via the Multicore Communication API (MCAPI), and for task management via the Multicore Task Management API (MTAPI). Since these APIs are system agnostic, they facilitate development of portable code, thus improving the feasibility of running an application on more than just one hardware platform.

This paper makes the following contributions:

- Presents a light-weight software stack based on MTAPI targeting resource-constrained heterogeneous embedded systems
- Showcases two open source MTAPI implementations[6]
- Evaluates the implementations using case studies on an embedded platform equipped with an ARM processor and a GPU

The paper does not aim to compare and contrast between the two implementations. Instead, the goal is to discuss how they can be used by software developers to port applications to heterogeneous multicore systems.

The rest of the paper is organized as follows: Section 2 discusses related work, and Section 3 gives an overview of MTAPI. The design and usage of MTAPI runtime systems is described in Section 4. Section 5 presents our experimental results, and Section 6 concludes with some ideas for future work.

[5] `https://www.multicore-association.org`

[6] UH-MTAPI and Siemens MTAPI: Libraries created by the University of Houston and Siemens, respectively.

2 Related Work

In this section, we discuss state-of-the-art parallel programming models for heterogeneous multicore systems from the task parallelism perspective.

OpenMP has been widely used in HPC for exploiting shared memory parallelism [5] and recently extended to support heterogeneous systems. OpenMP 3.1 ratified tasks, and task parallelism for multicore SoCs was implemented by several frameworks [4, 9, 14]. OpenMP 4.0 extended tasks to support dependencies evaluated in [11]—again using traditional shared memory-based architectures.

Other task-based approaches include Intel's TBB [17] that assigns tasks to multiple cores through a runtime library. As most frameworks, however, TBB targets desktop and server applications and is not designed for low-footprint embedded and heterogeneous systems.

Cilk [2] is a C language extensions originally developed by MIT for multithreaded parallel computing. While Cilk simplifies the implementation of task parallel applications, it only supports shared memory environments which limits its application to homogeneous systems.

OpenCL [18] is a standard designed for data parallel processing and used to program CPUs, GPUs, DSPs, etc. Although the standard can target multiple platforms, there is a steep learning curve making it a challenge to be adaptable. Moreover, achieving high performance often requires significant refactoring when switching between different hardware platforms.

OmpSs (OpenMP SuperScalar) [8] is a task-based programming model which exploits parallelism based on annotations using `pragmas`. OmpSs has been extended to many-core processors with accelerators such as multiple GPU systems. However, OmpSs needs special compiler support which hampers its usage in embedded systems.

StarPU [1] is a tasking API that allows developers to design applications in heterogeneous environments. StarPU's runtime schedules the tasks and corresponding data transfers among the CPU and GPUs. However, the necessary extension plug-in for GCC puts constraints on the deployment of StarPU to embedded systems with limited resources or bare-metal devices.

As discussed above, there are many approaches that explore task parallelism. However, they may not be best suited for embedded platforms which, unlike traditional platforms, often lack resources and sometimes do not even have an operating systems. Additionally, many embedded systems are subject to real-time constraints and forbid dynamic memory allocation during operation which is completely ignored by the approaches discussed above.

Our prior work in [20] uses MCAPI to establish communication through well-pipelined DMA protocols between Freescale's P4080 Power Architecture and the specialized RegEx Pattern Matching Engine (PME). We also created an abstraction layer for easy programmability by translating OpenMP to MRAPI [23, 24]. Designed and implemented in ANSI C, MCAPI and MRAPI do not require specific compiler support or user-defined language extensions.

3 MTAPI Overview

Figure 1 gives an overview of MTAPI. Applications can be developed by directly calling the MTAPI interface or via further abstraction layers such as OpenMP (a translation from OpenMP to MTAPI is described in [21]). MTAPI can be implemented on top of most operating systems and may even run bare metal to its simple design and minimal dependencies.

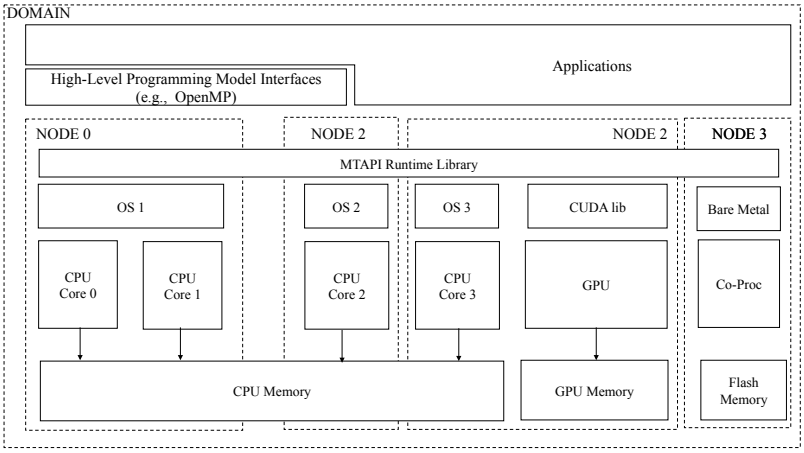


Fig. 1. MTAPI framework

In the following, we describe the main concepts of MTAPI.

Node: An MTAPI node is an independent unit of execution. A node can be a process, a thread, a thread pool, a general purpose processor or an accelerator.

Job and Action: A job is an abstraction representing the work and is implemented by one or more actions. For example, a job can be implemented by one action on the CPU and another action on the GPU. The MTAPI system binds tasks to the most suitable actions during runtime.

Task: An MTAPI task is an instance of a job together with its data environment. Since tasks are fine granular pieces of work, numerous tasks can be created, scheduled, and executed in parallel. A task can be offloaded to a neighboring node other than its origin node depending on the dynamic action binding. Therefore, optimized and efficient scheduling algorithms are desired for task management on heterogeneous multicore platforms.

Queue: A queue, as specified by the MTAPI standard, guarantees sequential execution of tasks.

Group: MTAPI groups are defined for synchronization purposes. A group is similar to a barrier in other task models. Tasks attached to the same group must be completed before the next step by calling `mtapi_group_wait`.

MTAPI has attained increasing attention in recent years. For example, the European Space Agency (ESA) created an MTAPI implementation [3] for a LEON4 processor, which is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architectures. Wallentowitz et al. [22] developed a baseline implementation and plans for deploying MCAPI and MTAPI on tiled many-core SoCs.

Siemens has developed an own industry-grade MTAPI implementation as part of a larger open source project called Embedded Multicore Building Blocks (EMB^2). EMB^2 has been specifically designed for embedded systems and the typical requirements that accompany them, such as predictable memory consumption, which is essential for safety-critical applications, and real-time capability. For the latter, the library supports task priorities and affinities, and the scheduling strategy can be optimized for non-functional requirements such as minimal latency and fairness.

Besides the task scheduler, EMB^2 provides parallel algorithms like loops and reductions, concurrent data structures, and high-level patterns for implementing stream processing applications. These building blocks are largely implemented in a non-blocking (lock-free) fashion, thus preventing frequently encountered pitfalls like lock contention, deadlocks, and priority inversion. As another advantage in real-time systems, the algorithms and data structures give certain progress guarantees [12].

In the following, we explore the usability and applicability of MTAPI for heterogeneous embedded multicore platforms focusing on the implementations from the University of Houston and Siemens.

4 MTAPI Design and Usage

4.1 Job Scheduling and Actions

As mentioned earlier, MTAPI decomposes computations into multiple tasks, schedules them among the available processing units, and combines the results after synchronization. Here, a task is defined as a light-weight operation that describes the job to be done. However, during task creation, the task does not know with which action it will be associated. MTAPI provides a dynamic binding policy between tasks and actions. This allows to schedule jobs on more than one hardware type, where the scheduler is responsible for balancing the load. Depending on where a task is located, it is either a local task (if it is implemented by an action residing on the same node) or a remote task. Figure 2 shows an example for the relationship between tasks and actions. In the example, Tasks 1 and 2 refer to Job A, which is implemented by Actions I and II on remote nodes. In contrast, Task 3 accomplished by Job B is executed on the same node as the application (Action III).

Listing 1.1 shows an implementation of a matrix multiplication using MTAPI. There are two action functions implementing the job: `ActionFunction_GPU` is implemented by a CUDA kernel and `ActionFunction_CPU` by a sequential CPU

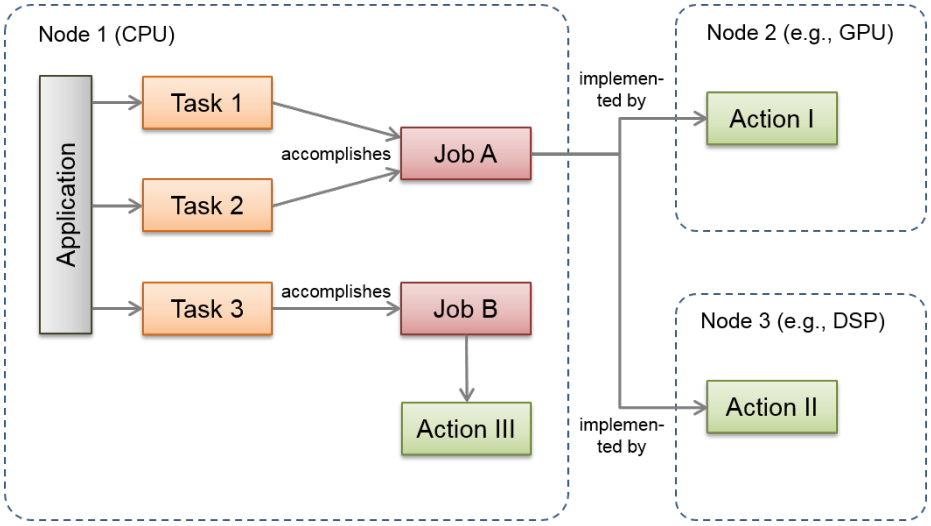


Fig. 2. MTAPI Tasks, Jobs, and Actions

kernel. After defining the two action functions, we attach them to the matrix multiplication job. Then, we create three tasks which are assigned to different actions in order to utilize both the GPU and the CPU.

4.2 Inter-Node Communication

For MTAPI users, communication between the nodes is completely transparent. However, this requires appropriate communication mechanisms in the runtime system. Siemens' MTAPI implementation builds on a plugin architecture which hides data transfers between the nodes in a heterogeneous system. In addition to predefined plugins for OpenCL, CUDA, and socket-based network communication, users can write their own plugins, e.g., for utilizing FPGAs or other specialized hardware.

UH-MTAPI uses MCAPI, the Multicore Communication API, for communication between the nodes. Essentially, each node has a receiver and a sender thread. These threads initialize the MCAPI environment and create MCAPI endpoints for message passing through appropriate function calls. They together compose the MCAPI communication layer between the nodes of a domain. Technically, the data transferred between MTAPI nodes is packed as MCAPI messages. MCAPI then transports these messages across the nodes for load balancing and synchronization. A message contains the domain ID, node ID, and port ID. Once a message is created, it is inserted into a central message queue on the node waiting for the sender to initiate the communication. Every message is assigned a priority. High priority messages such as action updates are inserted at the head of the message queue while low priority messages, e.g., for load balancing, are inserted at the tail of the queue. The sender wraps the MTAPI message into

```
void ActionFunction_GPU (
        const void* arguments,
        const mtapi_size_t arguments_size,
        void* result,
        const mtapi_size_t result_size,
        const void* node_local_data,
        const mtapi_size_t node_local_data_size,
        mtapi_task_context_t* const context)
{
        Argument_t* arg = (Argument_t*)arguments;
        CUDA_Kernel(arg->A, arg->B, arg->C, arg->n);
}

void ActionFunction_CPU (
        const void* arguments,
        const mtapi_size_t arguments_size,
        void* result,
        const mtapi_size_t result_size,
        const void* node_local_data,
        const mtapi_size_t node_local_data_size,
        mtapi_task_context_t* const context)
{
        Argument_t* arg = (Argument_t*)arguments;
        CPU_Kernel(arg->A, arg->B, arg->C, arg->n);
}

mtapi_action_create(JOB_MM, ActionFunction_CPU, NULL, 0,
        NULL, &status);
mtapi_action_create(JOB_MM, ActionFunction_GPU, NULL, 0,
        NULL, &status);
mtapi_task_hndl_t task[3];
task[0] = mtapi_task_start (0, JOB_MM, arg,
        sizeof(Argument_t), NULL, 0, NULL, group, &status);
task[1] = mtapi_task_start (0, JOB_MM, arg,
        sizeof(Argument_t), NULL, 0, NULL, group, &status);
task[2] = mtapi_task_start (0, JOB_MM, arg,
        sizeof(Argument_t), NULL, 0, NULL, group, &status);
mtapi_task_wait(task[0], MTAPI_INFINITE, &status);
mtapi_task_wait(task[1], MTAPI_INFINITE, &status);
mtapi_task_wait(task[2], MTAPI_INFINITE, &status);
```

Listing 1.1. MTAPI Matrix Multiplication

an MCAPI message according to its type and sends it to its destination node. The receiver thread keeps listening to its neighboring nodes to check if there is an MCAPI message sent to this node. Upon receipt of an MCAPI message, the receiver decodes it and creates an MTAPI message carrying the necessary information. Then, the receiver pushes the newly created MTAPI message into the message queue, waiting for the next cycle of message processing by the sender thread. Finally, the receiver thread continues listening to its neighboring nodes. In the UH-MTAPI design, a priority scheduler manages the message queue. The priority scheduler uses a centralized message queue, where the messages are sorted.

5 Performance Evaluation

In this section, we evaluate the Siemens MTAPI implementation[7] and UH-MTAPI[8]. We selected applications from *BOTS* [9] and *Rodinia* Benchmarks [6] to demonstrate their performance. The benchmarks are executed on NVIDIA's Jetson TK1 embedded development platform [15] with a Tegra K1 processor which integrates a 4-plus-1 ARM Cortex-A15 processor and a Kepler GPU with 192 cores. We use the GCC OpenMP implementation shipped with the board as reference for comparison purposes.

SparseLU Benchmark: The SparseLU factorization benchmark from *BOTS* computes an LU matrix factorization for sparse matrices. A sparse matrix contains submatrix blocks that may not be allocated. The vacancy of certain unallocated submatrix blocks leads to imbalance. Thus, task parallelism has better performance over other work sharing directives such as OpenMP's `parallel for`. In the SparseLU factorization, tasks are created only for the allocated submatrix blocks to reduce the overhead caused by imbalance.

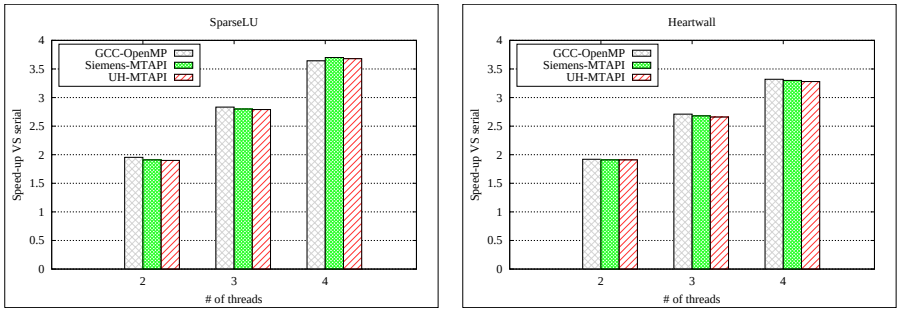
The sparse matrix contains 50×50 submatrices, where each submatrix has size 100×100 on both hardware platforms. We collect multiple metrics such as execution time, matrix size, and number of threads. The execution time for calculating the speed-up is measured on the CPU for the core part of the computation, excluding I/O and initial setup.

Figure 3(a) shows the speed-up using different implementations. The UH-MTAPI implementation demonstrates comparable performance with the Siemens MTAPI implementation as well as GCC's OpenMP version. Both Siemens and UH-MTAPI implementations achieve a roughly linear speed-up which indicates their scalability on multicore processors.

Heartwall benchmark: The Heartwall tracking benchmark is an application from *Rodinia* [6] which tracks the changing shapes of a mouse heart wall. We reorganized it by splitting loop parallelism into tasks, where each task handles a chunk of the image data. The image procedures are encapsulated in an action function that processes the data associated with the corresponding tasks.

[7] `https://github.com/siemens/embb`
[8] `https://github.com/MCAPro2015/OpenMP_MCA_Project`

Figure 3(b) shows the speed-up over a single thread. We observe that task parallelism conducted by UH-MTAPI matches the performance of data parallelism offered by OpenMP `parallel for` and the Siemens MTAPI implementation. However, none of the three versions meets the expectation of linear speedup as the number of threads increases.



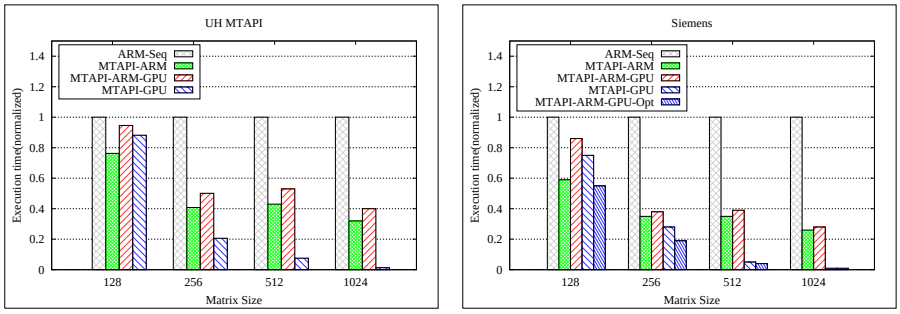
(a) SparseLU benchmark (b) Heartwall benchmark

Fig. 3. Speed-up for SparseLU and Heartwall benchmarks with OpenMP, Siemens MTAPI and UH-MTAPI on an NVIDIA Tegra TK1 board

Matrix-Matrix Multiplication: This benchmark (multiplication of dense matrices) is relatively compute-intensive. The complexity of a traditional multiplication of two square matrices is $\mathcal{O}(n^3)$. Although matrix multiplication can be implemented using the parallel working directives such as OpenMP's `parallel for`, the computation takes a lot of time due to the limited number of CPU threads and poor data locality. In contrast, heterogeneous systems with accelerators such as GPUs are a good fit for such algorithms, specifically as their architectures with a large amount of processing units allow to run many threads concurrently. Additionally, GPU matrix-matrix multiplication algorithms are potentially more cache friendly than CPU algorithms [10]. We implemented different types of action functions targeting the different processing units. The CPU action is implemented in C++ while the GPU action relies on CUDA [16]. Moreover, we designed four different approaches to execute the benchmark and an additional approach was used by Siemens to achieve maximum performance using both the CPU and the GPU:

ARM-Seq Sequential implementation on ARM CPU.
MTAPI-CPU MTAPI implementation with a single action for ARM CPU.
MTAPI-CPU-GPU MTAPI implementation with actions for CPU and GPU.
MTAPI-GPU MTAPI implementation with a single action for GPU.
MTAPI-CPU-GPU-Opt Same as MTAPI-CPU-GPU, but with work item sizes tailored to the particular needs of the respective computation units and copying of data to the GPU overlapped with computation.

Figure 4(a) shows the normalized execution times for matrix sizes 128, 256, 512, and 1024 for UH-MTAPI. Figure 4(b) shows the results for Siemens MTAPI. We observe that the ARM action has comparable performance with the GPU action for matrices with sizes less than 128.



(a) UH-MTAPI (b) Siemens MTAPI

Fig. 4. Normalized execution times for matrix multiplication with UH-MTAPI and Siemens MTAPI on an NVIDIA Tegra TK1 board

The data being copied between the CPU and the GPU poses a major communication overhead. However, as the matrix size increases, the data copying time can be largely ignored for which reason the GPU action outperforms the CPU action. A simple distribution of the work to both processing units did not yield a speedup. In fact, the CPU action is far slower than the GPU action, and equally sized work items let the GPU finish while the CPU is still calculating. For this reason, the optimized version uses bigger work items for the GPU and smaller ones for the CPU. Moreover, data is transferred asynchronously, thus hiding the transfer time in computations. This technique results in a speedup for all cases, but the contribution of the CPU shrinks with increasing matrix size as expected.

6 Conclusion and Future Work

Programming models for heterogeneous multicore systems are important yet challenging. In this paper, we described the design and implementation of a parallel programming standard, the Multicore Task Management API (MTAPI). MTAPI enables application-level task parallelism on embedded devices with symmetric or asymmetric multicore processors. We showed that MTAPI provides a transparent way to develop portable and scalable applications targeting heterogeneous systems. Our experimental results show that MTAPI implementations offer competitive performance compared to OpenMP while being more flexible. In the future, we will target further platforms such as DSPs and FPGAs.

Acknowledgments

Our sincere gratitude to the anonymous reviewers and many thanks to Markus Levy, President of the Multicore Association for his continued support.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. ACM SIGPLAN Notices 30(8) (1995)
3. Cederman, D., Hellström, D., Sherrill, J., Bloom, G., Patte, M., Zulianello, M.: RTEMS SMP for LEON3/LEON4 multi-processor devices. Data Systems in Aerospace (2014)
4. Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., Gatherer, A.: Implementing OpenMP on a high performance embedded multicore MPSoC. In: Parallel & Distributed Processing. pp. 1–8. IEEE (2009)
5. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: Portable shared memory parallel programming. MIT Press (2008)
6. Che, S., Boyer, M., Meng, J. et al.: Rodinia: A benchmark suite for heterogeneous computing. In: International Symposium on Workload Characterization (IISWC). pp. 44–54. IEEE (2009)
7. Dagum, L., Enon, R.: OpenMP: An industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5(1), 46–55 (1998)
8. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02), 173–193 (2011)
9. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP task scheduling strategies. In: OpenMP in a new era of parallelism, pp. 100–110. Springer (2008)
10. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: Proceedings of conference on graphics hardware. pp. 133–137. ACM (2004)
11. Ghosh, P., Yan, Y., Eachempati, D., Chapman, B.: A prototype implementation of OpenMP task dependency support. In: OpenMP in the Era of Low Power Devices and Accelerators, pp. 128–140. Springer (2013)
12. Herlihy, M., Shavit, N.: On the nature of progress. In: International Conference on Principles of Distributed Systems (OPODIS). pp. 313–328. Springer (2011)
13. Li, T., Brett, P., Knauerhase, R., Koufaty, D., Reddy, D., Hahn, S.: Operating system support for overlapping-ISA heterogeneous multi-core architectures. In: IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 1–12. IEEE (2010)
14. Liao, C., Hernandez, O., Chapman, B. et al.: OpenUH: An optimizing, portable OpenMP compiler. Concurrency and Computation: Practice and Experience 19(18), 2317–2332 (2007)
15. NVIDIA: Jetson TK1 development kit. `http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson_platform_brief_May2014.pdf`
16. NVIDIA: CUDA programming guide (2008)

17. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism. O'Reilly (2007)
18. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering 12(1-3), 66–73 (2010)
19. Stotzer, E., Jayaraj, A., Ali, M., Friedmann, A., Mitra, G., Rendell, A.P., Lintault, I.: OpenMP on the low-power TI keystone II ARM/DSP system-on-chip. In: OpenMP in the Era of Low Power Devices and Accelerators, pp. 114–127. Springer (2013)
20. Sun, P., Chandrasekaran, S., Chapman, B.: Targeting Heterogeneous SoCs using MCAPI. In: SRC TECHCON 2014, in the GRC Research Category Section 29.1 (2014), https://www.src.org/library/publication/p070788/
21. Sun, P., Chandrasekaran, S., Zhu, S., Chapman, B.: Deploying OpenMP task parallelism on multicore embedded systems with MCA task APIs. In: High Performance Computing and Communications (HPCC) (2015)
22. Wallentowitz, S., Wagner, P., Tempelmeier, M. et al.: Open tiled manycore system-on-chip. arXiv preprint arXiv:1304.5081 (2013)
23. Wang, C., Chandrasekaran, S., Chapman, B., Holt, J.: libEOMP: A portable OpenMP runtime library based on MCA APIs for embedded systems. In: Workshop on Programming Models and Applications for Multicores and Manycores (PMAM). pp. 83–92 (2013)
24. Wang, C., Chandrasekaran, S., Sun, P. et al.: Portable mapping of OpenMP to multicore embedded systems using MCA APIs. In: Proceedings of. pp. 153–162. Languages, Compilers, and Tools for Embedded Systems (LCTES) (2013)