OpenMP for Embedded Systems

Sunita Chandrasekaran
Asst. Professor
Dept. of Computer & Information Sciences
University of Delaware
schandra@udel.edu

ACK: Peng Sun, Suyang Zhu, Cheng Wang, Barbara Chapman, Tobias Schuele, Marcus Winter

Talk @ the OpenMP Booth #611, November 16, 2016
Heterogeneous Embedded Systems

• Incorporates specialized processing capabilities to handle specific tasks
• Example
  • CPU + GPU
  • ARM + GPU
  • ARM + DSP
  • CPU + FPGA
Programming Multicore Embedded Systems – A Real Challenge

• Heterogeneous systems present complexity at both silicon and system level
• Standards and tool-chain in embedded industry are proprietary
• Portability and scalability issues
• High time-to-market (TTM) solutions
• We need industry standards
  – To offer portable and scalable software solutions and target more than one platform
How suitable are the state-of-the-art models for heterogeneous embedded systems?

- Not portable across more than one type of platform except for OpenCL
- Most models are heavy-weight for embedded processors of limited resources
- Most models require support from OS and compilers
  - Sometimes embedded systems are bare-metal
- Some of the solutions are restricted to just the homogeneous environment
So what do we really need?

• Something that’s not too low-level
• Something light-weight
• Something that can target heterogeneous embedded platforms (beyond CPUs-GPUs)
• Something that can help speed time-to-market for products
• Last but not the least – we need industry standards
Using industry standards

- Two of them used for this work
  - OpenMP
    - (high-level, directive-based)
  - Multicore Association (MCA) APIs
    - (low-level, light-weight catered to embedded systems)
Briefly, on OpenMP Implementations

- **Directives implemented via code modification and insertion of runtime library calls**
  - Typical approach is outlining of code in parallel region
  - Or generation of micro tasks
- **Runtime library responsible for managing threads**
  - Scheduling loops
  - Scheduling tasks
  - Implementing synchronization
- **Implementation effort is reasonable**

<table>
<thead>
<tr>
<th>OpenMP Code</th>
<th>Translation</th>
</tr>
</thead>
</table>
| ```c
int main(void)
{
int a,b,c;
#pragma omp parallel
private(c)
do_sth(a,b,c);
return 0;
}``` | ```
_INT32 main()
{
int a,b,c;
/* microtask */
void __ompregion_main1()
{
_INT32 __mplocal_c;
/*shared variables are kept intact, substitute accesses to private variable*/
do_sth(a, b, __mplocal_c);
}
...``` |

Each compiler has custom runtime support. Quality of the runtime system has major impact on performance.
History of OpenMP*

- **1997**: In spring, 7 vendors and the DOE agree on the spelling of parallel loops and form the OpenMP Arb. By October, version 1.0 of the OpenMP specification for Fortran is released.
- **1998**: Minor modifications.
- **1999**: First hybrid applications with MPI* and OpenMP appear.
- **2000**: The merge of Fortran and C/C+ specifications begins.
- **2001**: Unified Fortran and C/C++. Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendor.
- **2002**: Incorporates task parallelism: A hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.
- **2003**: Support min/max reductions in C/ C++.
- **2004**: Incorporates the group of OpenMP users, is formed and organizes workshops on OpenMP in North America, Europe, and Asia.
- **2005**: Major modification.
- **2006**: First hybrid applications with MPI* and OpenMP appear.
- **2007**: The merge of Fortran and C/C+ specifications begins.
- **2008**: Incorporates task parallelism: A hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.
- **2009**: Support min/max reductions in C/ C++.
- **2010**: First International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendor.
- **2011**: Incorporates task parallelism. A hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.
- **2012**: The merge of Fortran and C/C+ specifications begins.
- **2013**: Supports offloading execution to accelerator and coprocessor devices, SIMD parallelism, and more. Expands OpenMP beyond traditional boundaries.
- **2014**: Unified Fortran and C/C++: Bigger than both individual specifications combined.
- **2015**: OpenMP supports task loops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.
- **2016**: Incorporates task parallelism. A hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.
- **2017**: Support min/max reductions in C/ C++.
- **2018**: OpenMP supports task loops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.
Multicore Association APIs (MCA)

• Develops standards to reduce complexity involved in writing software for multicore chips
• Capturing the basic elements and abstract hardware and system resources
• Cohesive set of foundation APIs
  - Standardize communication (MCAPI)
  - Resource Sharing (MRAPI)
  - Task Management (MTAPI)
Multicore Task Management API (MTAPI)

- **MTAPI**
  - **Standardized API** for task-parallel programming on a wide range of hardware architectures
  - Developed and driven by practitioners of market-leading companies
  - Part of Multicore-Association's **ecosystem** (MRAPI, MCAPI, SHIM, OpenAMP, …)

- **Ack:** Siemens (Tobias Schuele, Urs Gleim)
OpenMP and MCA software stack
Example for the usage of MTAPI in heterogeneous systems:

Node 1 (CPU):
- Task 1
- Task 2
- Task 3

Matrix mult.

Job A

FFT

Job B

Action III

Node 2 (GPU):
- Action I

Node 3 (DSP):
- Action II

Acknowledgement: Siemens (Tobias Schuele, Urs Gleim)
MTAPI implementations

Embedded Multicore Building Blocks (EMB2)\(^1\)
- Open source library and runtime platform for embedded multicore systems
- Real-time capability, resource awareness
- Fine-grained control over core usage (task priorities, affinities)

MTAPI implementation developed at the Universities of Houston / Delaware\(^2\)
- Utilizes MCAPI for inter-node communication and MRAPI for resource management
- Used as runtime system for OpenMP programs

---

\(^1\) [github.com/siemens/embb](https://github.com/siemens/embb)
\(^2\) [github.com/MCAPro2015/OpenMP_MCA_Project](https://github.com/MCAPro2015/OpenMP_MCA_Project)
Example for scheduling MTAPI tasks in heterogeneous systems:

Node 0 (CPU)
- Core 0
  - Worker thread 0
    - Q_{00}
    - Q_{01}
    - Q_{02}
- Core 1
  - Worker thread 1
    - Q_{10}
    - Q_{11}
    - Q_{12}

Node 1 (DSP)
- Unit 0
  - Q_{0}

Scheduler
- Work Stealing

Work stealing

Ack: Siemens (Tobias Schuele, Urs Gleim)
Testbed, Compiler and Benchmark

- Test beds:
  - NVIDIA Jetson TK1 embedded development board with a Tegra K1 Soc
    (NVIDIA 4-Plus-1 Quad-Core ARM Cortex-A15 processor and a Kepler GPU
    with 192 CUDA cores).
- Compiler: Jetson (GCC 4.8.4, NVCC V6.5.30)
- Power Architecture from Freescale
  - Consisting of Pattern Matching Engine as specialized accelerator
- Benchmarks: ¹Rodinia and ²BOTS.
- Reference Group: ³Siemens MTAPI, GCC OpenMP

Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators
²BOTS: https://pm.bsc.es/projects/bots
³Siemens-MTAPI: https://github.com/siemens/embb
Normalized execution times for UH-MTAPI and Siemens MTAPI (EMB²) for MM

Normalized execution times for UH-MTAPI and Siemens MTAPI (EMB²):

(a) UH-MTAPI
- MTAPI-ARM faster than MTAPI-GPU for small matrices due to overhead for data copying
- MTAPI-GPU faster than MTAPI-ARM-GPU for larger matrices due to load imbalance
- MTAPI-ARM-GPU-Opt always fastest due to asynchronous transfers and variable block sizes

(b) Siemens MTAPI
OpenMP RTL translation to MTAPI

- Compiler front end translates OpenMP constructs to MTAPI-RTL functions
- RTL comprises of MTAPI function calls and we convert OpenMP tasks to MTAPI objects
- Embedded resources will rely on MTAPI for management of resources

Diagram:
- OpenMP APP
  - Compiler
  - IR
  - Code Generator
  - MTAPI RTL
    - OpenMP-MTAPI RTL
      - MTAPI Tasks
        - CPU Binary
          - Linker
            - Executable

- OpenMP programs contain task construct
- OpenMP-MTAPI RTL includes the runtime calls of the translated task construct
- OpenMP-MTAPI RTL incurs and dispatches MTAPI tasks
OpenMP -> MTAPI Implementation
SparseLU
Takeaways and Summary

• Industry standards are the way to go!
• OpenMP-MCA incurred little to no overhead
  – Targeting heterogeneous platforms
• Less learning curve
• Ability to maintain single code base