

Parallel Programming Models on Microcontrollers

Niccolò Nicolosi, Daniele Cattaneo, Giovanni Agosta, Federico Terraneo
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
name.surname@polimi.it

Abstract—To overcome the programmability issue in multicore microcontrollers and to allow seamless portability of parallel programming libraries across the embedded computing continuum, in this paper we discuss the modifications required to the operating system and runtimes to support OpenMP and SYCL on microcontrollers. We compare the performance of parallel OpenMP and SYCL code on an emerging embedded kernel architecture: the fluid kernel. Experiments running on the RP2040 dual-core microcontroller using standard benchmarks achieve an average speedup of 1.64x (maximum 1.90x) for SYCL and 1.65x (maximum 1.89x) for OpenMP compared to the equivalent single-core code. To the best of our knowledge, we present the first results of running SYCL code on microcontrollers.

Index Terms—SYCL, OpenMP, embedded systems, microcontrollers, real-time operating systems

We acknowledge financial support under the National Recovery and Resilience Plan (NRPP), call for tender No. 104 published on 02/02/2022 by the Italian Ministry of University and Research (MUR), funded by the European Union - Next Generation EU, Mission 4, Component 1, CUP D53D23008590001 and D53D23008600006, project title LibreRT.

I. INTRODUCTION AND RELATED WORK

In traditional single-core microcontrollers, Real-Time Operating Systems (RTOSes) enable the use of threads for the purpose of multi-tasking and temporal isolation between real-time and non-real-time tasks. The steadily increasing algorithmic complexity of embedded software is however pushing the semiconductor industry to deliver higher and higher performance in microcontrollers. As a result, just like in High-Performance computing (HPC) and desktop computing, multicore microcontrollers are expected to soon become prevalent, and thus leveraging parallelism becomes a new relevant use of multi-threading in this field. Therefore, embedded programmers will increasingly need programming models facilitating algorithm parallelization.

The major programming models for parallelization are OpenMP, OpenCL, CUDA and SYCL [1], [2]. However, CUDA and OpenCL are focused on SIMD parallelism as found in GPUs, and are thus unsuitable for multicore microcontrollers which exploit CPU-based MIMD parallelism.

OpenMP is the dominant parallel programming API for multicore CPUs [3], available as a pragma-based extension of C, C++ or Fortran. The reliance on pragmas forces OpenMP to be implemented at compiler-level, and the underlying runtime calls emitted by each compiler (LLVM or GCC) are not standardized. OpenMP has been extended to support heterogeneous systems [4], [5], and Linux-capable high-end embedded systems with Memory Management Units (MMU) [6]–[8]. OpenMP has also been ported to the Texas Instruments

TMS320C64x+ [9] Digital Signal Processor (DSP). However, the authors do not explore multicore-aware RTOSes, opting to run separate OS instances on each DSP core instead.

On the other hand, SYCL targets both CPUs and GPUs and is increasingly popular in edge-computing applications. It is a single-source parallel programming model based on C++, that does not require language extensions such as OpenMP’s pragmas. As a result, SYCL support can be implemented as library-only [10]–[12], with compiler support only necessary when targeting GPUs. Multiple implementations of SYCL exist, such as AdaptiveCpp and DPC++ [13]. SYCL has been used in HPC for both CPU and GPU parallelism, and even for programming clusters of accelerators [14]. However, its adoption in the embedded world is currently in its infancy. There is a group working on the standardization of SYCL in safety critical applications [15], but its current focus is high-end heterogeneous embedded systems based on Linux. To the best of the authors’ knowledge, we are the first to target SYCL for microcontrollers and RTOSes.

While microcontrollers have traditionally been programmed using *unikernels*, which provide no separation between kernelspace and userspace code, the rise in complexity of modern microcontroller hardware promotes support for high-level programming models, like POSIX compliance and the abstraction of processes. This scenario has driven the recent introduction of embedded Real-Time Operating System (RTOS) architectures such as the fluid kernel [16]. Fluid kernels unify the embedded computing continuum by enabling the use of standard APIs also supported by Linux in RTOSes, supporting both kernel-space applications (*unikernel*-like) and user-space applications through processes. Therefore it is a relevant research challenge to extend other high-level APIs standard such as parallel programming models to microcontrollers.

In this work we introduce SYCL and OpenMP support on MMU-less multicore microcontrollers running a fluid kernel RTOS, Miosix¹, with the goal to allow writing parallel code using standard programming models across the whole embedded computing continuum.

II. PARALLEL PROGRAMMING ON MICROCONTROLLERS

Multicore microcontroller support in RTOSes, despite being envisioned as far back as 2005 [17], is only becoming relevant now that Symmetric MultiProcessing (SMP) microcontrollers are commonly available. These microcontrollers are designed to run a single instance of an operating system on all the available cores, and provide the required hardware support to do so efficiently. A multicore-aware RTOS provides the

¹<https://github.com/fedetft/miosix-kernel>

```

1 //Full code in:
2 //kernel/scheduler/priority/priority_scheduler.cpp
3 Thread *      running[CPU_NUM_CORES]; //global
4 FifoQueue<Thread *> ready[NUM_PRIORITIES]; //global
5 List<Thread *> notReady; //global
6
7 void scheduler() {
8     // Deschedule previous thread
9     auto prev=running[getCoreId()];
10    if(prev->isReady()==false) notReady.push_back(prev);
11    else ready[prev->getPriority()].push_back(prev);
12    // Find next thread to run
13    int i=NUM_PRIORITIES-1;
14    for (;i>=0;i--)
15        if(ready[i].empty()==false) {
16            running[getCoreId()]=ready[i].front();
17            ready[i].pop_front();
18            computePreemption(i);
19            break;
20        }
21    // Check other cores schedule
22    int minRunPrio, coreIdRunMinPrio;
23    tie(minRunPrio,coreIdRunMinPrio)=minPriority(running);
24    for (;i>minRunPrio;i--)
25        if(ready[i].empty()==false) {
26            invokeSchedulerOnCore(coreIdRunMinPrio);
27            break;
28        }
29 }

```

Listing 1. Miosix simplified scheduler algorithm.

required scheduling and synchronization primitives for exploiting shared memory, which is the prevalent paradigm used for parallel application code.

A. Adding multicore support to the Miosix fluid kernel

Originally, Miosix used a priority-based preemptive scheduler [18], enforcing the hard real-time invariant that the ready thread with the highest priority should always be executing. If multiple threads with the same priority exist, they are scheduled in round-robin fashion with a configurable preemption time quantum. The multicore scheduler enforces the same invariant (it guarantees no core is running a thread with lower priority than one waiting in the ready queue) using a distributed approach. The scheduler’s algorithm is shown in Listing 1. Each scheduler invocation only updates the running thread on the current core (lines 8-20 in the listing), and then checks that the schedule is globally respecting the invariant, invoking itself on another core if needed (lines 22-28). Note that although the scheduling algorithm is distributed, it operates on a shared ready queue (the global array *ready*). This queue is protected by a spinlock (not shown in the listing) while the scheduler is in execution.

For implementing synchronization primitives, current microcontrollers provide a fixed number of hardware spinlocks (32 in the RP2040). Miosix uses them to implement two global locks, the Global Interrupt disable Lock (GIL) and the Global Preemption disable Lock (GPL). The GIL protects threads or interrupt handlers from other threads or interrupt handlers, and is used by the scheduler and device drivers. The GPL only protects threads from other threads, and is used to implement POSIX and C++11 mutexes and *condition_variables*, which ensure that a waiting thread is put to sleep. These objects cannot be implemented as hardware spinlocks to

prevent, for instance, starvation issues if a waiting thread is preempted [19].

B. OpenMP & SYCL Support for Miosix

OpenMP and SYCL implementations were originally developed for HPC, thus not taking into consideration the limitations of embedded hardware. However, once multicore support for POSIX threads was added to Miosix, we found out that existing parallel runtimes can run on microcontrollers with minimal modifications. Specifically, we performed optimizations to reduce memory usage, and removed the use of thread-local storage (currently not supported on Miosix). For what concerns OpenMP, we adapted both the libgomp [20] runtime for GCC, and the libomp [21] one for LLVM clang, in the versions shipped together with GCC 9.2.0 and LLVM 18.1.8 respectively. For what concerns SYCL, C++ support is also needed, which for Miosix was not a concern as the kernel provides native support for the C++ standard library on microcontrollers. Amongst several SYCL runtime alternatives we chose AdaptiveCpp [22] for its library-only CPU-based implementation which currently uses OpenMP as backend. This choice allowed us to reuse the work already done for OpenMP. The runtimes, once adapted, had a resulting memory overhead of 8KB for libgomp, 14KB for libomp, 11KB for AdaptiveCpp compiled with GCC/libgomp, and 23KB for AdaptiveCpp compiled with LLVM clang/libomp. Our modifications to the OpenMP and SYCL runtimes are available in this repository: https://github.com/HEAPLab/embedded_parallel_programming.

III. EXPERIMENTAL EVALUATION

To measure the effectiveness of OpenMP and SYCL on microcontrollers, we selected two benchmark suites: PolyBench/ACC [23] for OpenMP, and SYCL-bench [13] for SYCL. PolyBench/ACC is a version of PolyBench/C [24] accelerated with OpenMP. SYCL-bench is a standard for SYCL benchmarking, and it is based on a port of PolyBench/ACC to SYCL, allowing a direct comparison between the suites.

A. Experimental Setup

The experimental evaluation in this paper has been performed on the RP2040 [25] microcontroller from Raspberry Pi, whose architecture is outlined in Figure 1. This microcontroller is based around two ARM Cortex-M0+ cores running at

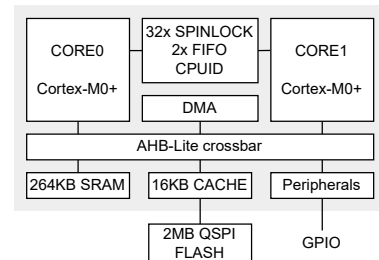


Fig. 1. Architecture of the RP2040 microcontroller.

200MHz. Each core has one 32 bit memory access port, shared for both instruction and data accesses. A separate 32 bit port, mapped to a subset of the address space, is connected to a shared device providing hardware spinlocks and FIFO queues for inter-core synchronization, as well as a CPUID register allowing code to know on which core it is currently running on. The microcontroller provides an on-chip 264KB single-cycle SRAM divided in 6 striped banks. Cores can access different banks in the same clock cycle, while a round-robin arbiter is used when both cores attempt to access the same bank. Code execution is performed directly from an off-chip FLASH memory (eXecute In Place or XIP) connected through a 4 bit QSPI bus running at 100MHz. An on-chip 16KB cache, shared between cores, is used to compensate for the slow external memory speed. The code for all benchmarks is available at https://github.com/HEAPLab/embedded_parallel_programming.

On the software side, the code for the benchmarks we are using for the evaluation (PolyBench/ACC² and SYCL-bench³) was left unmodified in the parts related to the kernels themselves and the dataset initialization code. Only the sizes of the datasets were adjusted in order to fit in the RP2040's available RAM memory, while keeping the same sizes for both PolyBench/ACC and SYCL-bench. The speedups were computed by comparing the execution times with the ones obtained by forcing the same benchmarks to run on a single core only. We measure the execution times through the facilities built-in to the benchmark suites. PolyBench/ACC uses `gettimeofday` while SYCL-bench employs the C++ `chrono` API. On Miosix, both return a sub-microsecond-resolution time value. In the experiments we skipped benchmarks that did not exploit CPU parallelization. Additionally, some of the OpenMP benchmarks contained illegal pragmas for the latest version of the standard. These benchmarks were likewise skipped to avoid modifying their code, which would impair reproducibility. No outdated code was present in SYCL-bench, but only the benchmarks that were also part of PolyBench/ACC were tested. All the benchmarks were compiled with GCC 9.2.0 and LLVM clang 18.1.8 both, the two compilers supported by Miosix. This allows to compare the OpenMP implementations between GCC and LLVM.

B. Sequential vs. parallel run times

In Figure 2 and in Figure 3 the bars show the speedups obtained through OpenMP and SYCL respectively on the targeted subsets of PolyBench benchmarks on the RP2040. Both for OpenMP and SYCL we see a maximum speedup up to $1.9\times$, very close but lower than the theoretical maximum of $2\times$ for a two core architecture. This is due to synchronization overhead and intrinsic contention on the AHB-Lite crossbar due to the lack of per-core caches. For the same benchmarks, we see about the same speedups using either GCC or LLVM clang, with GCC being slightly more performant overall. We believe this is due to the GCC OpenMP runtime being more lightweight than the LLVM one. The speedups are also

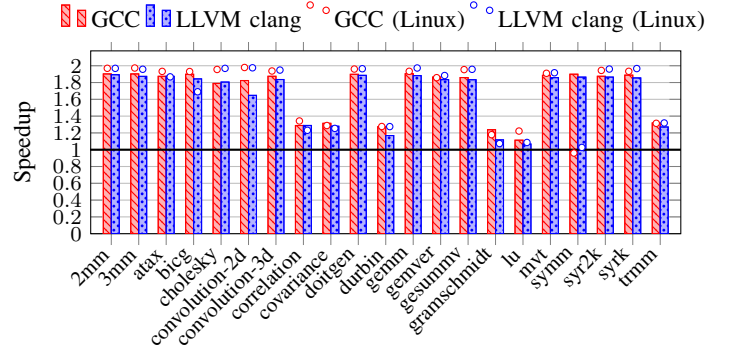


Fig. 2. Speedup of PolyBench/ACC benchmarks using OpenMP on the RP2040 compared to serial (single core) execution. The dots represent the speedups obtained on an Intel Core i7-9700K running Linux.

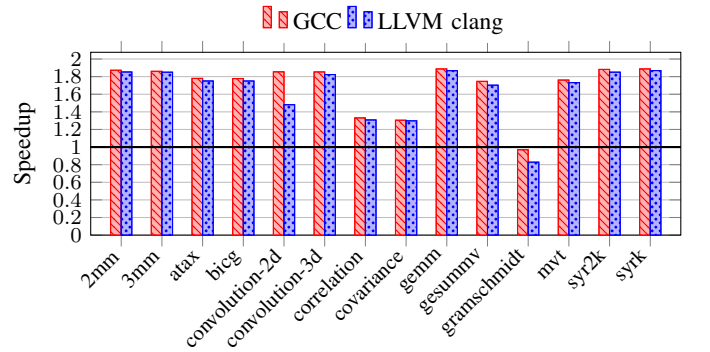


Fig. 3. Speedup of SYCL-bench benchmarks on the RP2040 compared to serial (single core) execution.

consistent between SYCL and OpenMP, highlighting that the per-thread overhead of SYCL is not much greater than the one of OpenMP, despite the targeted implementation of SYCL being based on OpenMP.

Some benchmarks exhibit poor speedup, in particular *correlation*, *covariance*, *durbin*, *gramschmidt*, *lu* and *trmm*. This is due to a limitation in how the benchmarks are written, which poorly partitions the work amongst the available cores. The partitioning can be significantly improved by using the `schedule(static, 2)` OpenMP pragma in the inner loops of the affected benchmarks, although we opted to present results for unmodified benchmarks. Poor speedups on the same benchmarks can be reproduced on a desktop platform, independently of the data size. For instance, on a Intel Core i7-9700K CPU running at 4.8GHz, running Ubuntu Linux 24.04 LTS, using OpenMP limited to two threads, we see similar speedups as with Miosix (dots in Figure 3), with the only exception of *symm*. Considering that the inner loop of *symm* accumulates over the same elements of an array in all worker threads, the lack of a speedup is attributable to cache coherency effects [26] which are absent on the RP2040.

C. Energy measurements

We measured the energy consumption during the execution of the benchmarks in order to evaluate the impact of using

²<https://cavazos-lab.github.io/PolyBench-ACC>

³<https://github.com/unisa-hpc/sycl-bench>

both cores. The measurement was performed using a Teledyne T3DSO1104 oscilloscope, measuring the current draw as sensed by a 0.01Ω shunt resistor pre-amplified by a factor of $50\times$. The resulting trace, multiplied by the supply voltage, results in a power trace. Integrating the power trace within the time frame when each benchmark was executing results in the benchmark's energy measurement. We observed that the increased power consumption due to parallelization on two cores is compensated by the reduced time-to-solution, resulting in comparable energy consumption for the single-core and parallel executions. Indeed, the greatest difference in power consumption across all benchmarks was a 6.92% increase for the parallel execution of *syrk* with SYCL using LLVM clang, which amounts to a difference of just 0.174J in absolute terms. Similar maximum figures were obtained for the other configurations: a 5.95% energy increase with SYCL using GCC, still running *syrk*; a 5.66% increase with OpenMP using LLVM clang, and 4.02% increase with SYCL using GCC running *syrk*.

D. Co-scheduling with real time threads

To demonstrate the possibility to run code making use of parallel programming models alongside real-time application in the same hardware and operating system, we co-scheduled parallelized OpenMP and SYCL code with real-time threads. The real-time code is composed of two high-priority threads, each running a periodic computation occupying the CPU for a workload of 7ms with a period of 10ms, the period being equal to the deadline. Since the total real-time workload is 140%, it is not schedulable on a single core architecture. It is, however schedulable on the RP2040 dual-code microcontroller. In addition to the two high-priority threads, the OpenMP and SYCL *2mm* benchmark was run concurrently utilizing the remaining CPU time. In both cases no deadline misses were observed.

IV. CONCLUSION AND FUTURE WORKS

In this paper, we explored the possibility to run parallel programming models on multicore microcontrollers. We discussed the multicore extension to the Miosix fluid kernel, and the modifications to existing parallel runtimes to utilize multithreading for exploiting parallelism in the underlying computer architecture. Thanks to the POSIX complicity of fluid kernels, the SYCL and OpenMP runtimes could run with minimal modifications on the RP2040 platform, showing that the industrial adoption of these technologies on microcontrollers is within reach. As shown by the achieved speedup of up to $1.9\times$ and the minimal energy overhead compared to single-core code, it is practical to run parallel SYCL and OpenMP code on microcontrollers. Thanks to preserving the same hard real-time invariant of the previous Miosix single-core scheduler, it is also possible to co-schedule higher priority real-time threads together with compute-intensive batch threads making use of parallel programming models. Future directions include providing extensions to said parallel programming models in order to make OpenMP and SYCL suitable for writing real-time code (as opposed to be run alongside existing real-time code).

ACKNOWLEDGMENT

No AI tools were employed in the writing of this paper.

REFERENCES

- [1] J. Fang *et al.*, "Parallel programming models for heterogeneous many-cores: a comprehensive survey," *CCF Transactions on High Performance Computing*, vol. 2, no. 4, pp. 382–400, 2020.
- [2] J. Diaz *et al.*, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.
- [3] T. Kadosh *et al.*, "Quantifying OpenMP: Statistical insights into usage and adoption," in *HPEC 2023*. [Online]. Available: doi.org/10.1109/HPEC58863.2023.10363459
- [4] E. Ayguadé *et al.*, "Extending OpenMP to survive the heterogeneous multi-core era," *Int. J. Parallel Program.*, vol. 38, pp. 440–459, 2010.
- [5] Martineau *et al.*, "Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model," in *IPDPSW 2016*, pp. 338–347.
- [6] P. Sun *et al.*, "OpenMP-MCA: Leveraging multiprocessor embedded systems using industry standards," in *IPDPS 2015*, pp. 679–688.
- [7] R. E. Vargas *et al.*, "A lightweight OpenMP4 run-time for embedded systems," in *ASP-DAC 2016*, p. 43–49.
- [8] B. D. de Dinechin, "Kalray MPPA@: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA manycore processor," in *HCS 2015*. [Online]. Available: doi.org/10.1109/HOTCHIPS.2015.7477332
- [9] B. Chapman *et al.*, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *IPDPS 2009*. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161107
- [10] A. Alpay *et al.*, "How much SYCL does a compiler need? Experiences from the implementation of SYCL as a library for nvc++," in *IWOCL 2022*. [Online]. Available: doi.org/10.1145/3529538.3529556
- [11] P. Thoman *et al.*, "Simsycl: A SYCL implementation targeting development, debugging, simulation and conformance," in *IWOCL 2024*. [Online]. Available: doi.org/10.1145/3648115.3648136
- [12] R. Reyes *et al.*, "SYCL: Single-source C++ accelerator programming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.
- [13] L. Crisci *et al.*, "SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs," in *IWOCL 2024*. [Online]. Available: doi.org/10.1145/3648115.3648120
- [14] P. Thoman *et al.*, "Celerity: High-level C++ for accelerator clusters," in *Euro-Par 2019: Parallel Processing*. Cham: Springer International Publishing, 2019, pp. 291–303.
- [15] E. Tomusk *et al.*, "What's new in SYCL for safety critical systems," in *IWOCL 2023*. [Online]. Available: https://doi.org/10.1145/3585341.3585367
- [16] F. Terraneo and D. Cattaneo, "Fluid kernels: Seamlessly conquering the embedded computing continuum," *IEEE Transactions on Computers*, vol. 74, no. 12, pp. 4050–4064, 2025.
- [17] "Multi-core microprocessors in embedded applications," https://www.nxp.com/docs/en/white-paper/multicoreWP.pdf, Accessed: 2025-07-07.
- [18] C. L. Liu *et al.*, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, p. 46–61, Jan. 1973.
- [19] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," in *SOSP '89*, p. 159–166. [Online]. Available: https://doi.org/10.1145/74850.74866
- [20] "GNU Offloading and Multi-Processing Project (GOMP)," https://gcc.gnu.org/projects/gomp/, Accessed: 2025-07-07.
- [21] "LLVM/OpenMP Runtimes," https://openmp.llvm.org/design/Runtimes.html, Accessed: 2025-07-07.
- [22] A. Alpay *et al.*, "AdaptiveCpp Stdpar: C++ standard parallelism integrated into a SYCL compiler," in *IWOCL 2024*. [Online]. Available: https://doi.org/10.1145/3648115.3648117
- [23] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to GPU codes," in *InPar 2012*. [Online]. Available: doi.org/10.1109/InPar.2012.6339595
- [24] L.-N. Pouchet and T. Yuki, "Polybench/C 4.2.1," 2016. [Online]. Available: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
- [25] "RP2040 datasheet," https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf, Accessed: 2025-07-07.
- [26] D. Molka *et al.*, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *PACT 2009*. [Online]. Available: https://doi.org/10.1109/PACT.2009.22