OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip

Eric Stotzer[1], Ajay Jayaraj[1], Murtaza Ali[1], Arnon Friedmann[1], Gaurav Mitra[2], Alistair P. Rendell[2], and Théa-Martine Gauthier[3]

[1] Texas Instruments, Dallas TX, USA
{estotzer,ajayj,mali,arnon}@ti.com
[2] Australian National University, Canberra ACT, Australia
{gaurav.mitra,alistair.rendell}@anu.edu.au
[3] nCore HPC, USA
thea@ncorehpc.com

Abstract. The Texas Instrument (TI) Keystone II architecture integrates an octa-core C66X DSP with a quad-core ARM Cortex A15 MPCore processor in a non-cache coherent shared memory environment. This System-on-a-Chip (SoC) offers very high Floating Point Operations per second (FLOPS) per Watt, if used efficiently. This paper reports an initial attempt at developing a bare-metal OpenMP runtime for the C66X multi-core DSP using the Open Event Machine RTOS. It also outlines an extension to OpenMP that allows code to run across both the ARM and the DSP cores simultaneously. Preliminary performance data for OpenMP constructs running on the ARM and DSP parts of the SoC are given and compared with other current processors.

1 Introduction

High performance computing has evolved to use specialized accelerators such as Graphics Processing Units (GPUs) for a variety of problems. However, such accelerators suffer from two main issues of excessive power consumption and insufficient device memory. Low-power SoCs with on-chip accelerators sharing the same address space and physical memory are increasingly being considered as alternatives. In recent work on using ARM NEON Floating Point Units (FPU) to accelerate application codes [1], low-power ARM based SoCs demonstrated comparable performance speedups to Intel processors using SSE2.

It has also been demonstrated that the TI Keystone I C66X Multi-core DSP provides higher GFLOPS/Watt (with 57% utilization of resources) for SGEMM matrix multiplication using OpenMP than Intel Core i7-960, IBM Cell Broadband Engine, Stratix IV FPGA and NVIDIA GTX480, GTX280 systems [2]. Increase in net utilization of the C66x DSP for other applications would result in higher GFLOPS/Watt. The TI Keystone II architecture integrates this C66X octa-core DSP with a quad-core ARM Cortex A15 MPCore processor. This combination of ARM and DSP processors on the same SoC promises excellent energy efficient performance if used efficiently.

Performance of OpenMP based applications depends heavily on the runtime library implementation. The runtime in [2] used the underlying SYS/BIOS RTOS. In this paper, we demonstrate a lighter weight OpenMP runtime implementation for the C66X multi-core DSP using the Open Event Machine RTOS. For key OpenMP directives, overheads are 2.5× lower than the previous implementation using SYS/BIOS. EPCC v3 micro-benchmarks [3] are provided for Keystone II and other Intel, ARM processors for comparison.

The rest of the paper is organized as follows. Section 2 provides a concise overview of the TI Keystone architecture. Description of our new bare-metal implementation of the OpenMP runtime for C66X DSP is given in Section 3. A brief introduction to our OpenMP accelerator dispatch prototype is outlined in Section 4. Micro-benchmarks of CPU cycle overheads for OpenMP constructs are discussed in Section 5. Related work, conclusions and future work, and acknowledgements follow in Sections 6, 7 and 7.

2 TI Keystone Overview

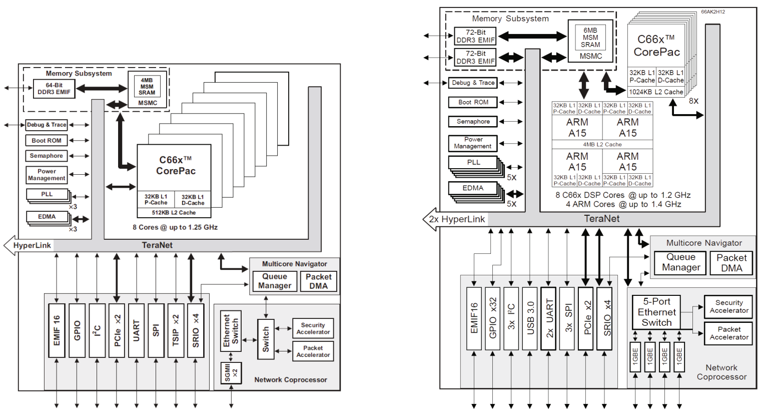
The Keystone architecture from Texas Instruments is an innovative platform integrating RISC and DSP cores along with application-specific co-processors and input/output peripherals. This high performance structure includes adequate internal bandwidth for non-blocking access to all processing cores, peripherals, co-processors and I/O. Figure 1 shows two instantiations of the Keystone architecture applicable to high performance compute applications.

2.1 C66x DSP Core

The main compute core inside the Keystone architecture is the C66x DSP from Texas Instruments [4]. This is based on a Very Long Instruction Word (VLIW) architecture. The core has two data-paths, each capable of executing four instructions per cycle on four functional units named M, D, L and S. The M unit primarily performs multiplication operations, the D-unit performs load/store and address calculations, and the L and S units perform addition and logical operations. Overall the two data-paths appear as an 8-way VLIW machine capable of executing up to eight instructions in each cycle. The instruction set also includes Single Instruction Multiple Data (SIMD) instructions allowing vector processing on up to 128-bit vectors. For example, the M unit can perform four single precision multiplies per cycles, whereas each L and S unit can each perform two single precision additions per cycle. Together the two data-paths can issue 16 single FLOP per cycle. The double precision capability is about one-fourth of single precision FLOPs.

2.2 C6678 'Shannon' System-on-Chip

The C6678 System-on-Chip (SoC) is the highest performance Keystone I device that includes only DSP cores [5]. Figure 1(a) shows the block diagram of



(a) C6678 SoC based on Keystone I Architecture (b) 66AK2H12 SoC based on Keystone II Architecture

Fig. 1. Texas Instruments Keystone Architectures

this device. It has eight C66x cores and a three-level memory system. The cores can run at 1.25 GHz, thereby providing a peak performance of 160 single precision GFLOPS and 40 double precision GFLOPS. The memory system is a Non-Uniform Memory Architecture (NUMA) [6]. A C66x subsystem can access different memory regions, with accesses to memories that are physically closer to a processor being faster. The memory regions and access times are as follows:

- Level-1 program (L1P) and data (L1D): 32KB, 1-cycle access time, configurable as mapped RAM, cache, or a combination of mapped and cached.
- Local-L2: 512KB, 2-cycle access time, configurable as mapped RAM, cache, or a combination of mapped and cached, and shared between the L1D and L1P caches.
- Shared-L2: 4MB, 2-cycle access time, shared memory on-chip.
- Shared-L3: multiple megabytes of off-chip memory with greater than 60-cycle access time. One DDR3 controller in this device.

The device comes with a rich set of standard interfaces like PCI express, Serial Rapid I/O (SRIO), multiple Gigabit Ethernet ports as well as a proprietary interface known as the Hyperlink that provides a 50 Gbps point-to-point connectivity.

2.3 66AK2H12 'Hawking' System-on-Chip

The 66AK2H12 SoC is the highest performance Keystone II device architecture that includes an ARM RISC processor in addition to the compute-efficient C66x DSP cores [7]. This particular device (Figure 1(b)) integrates a Cortex-A15 quad-core cluster and a C66x DSP octa-core cluster. The Cortex-A15 quad cores are fully cache coherent, although as on the C6678 the DSP cores do not maintain cache coherency. External memory bandwidth is doubled with dual DDR3 controllers. An additional Hyperlink interface is also included. Compared to the

Keystone I C6678 SoC the memory sizes are also increased. On the DSP, the L1D and L1P cache sizes remain at 32KB per core, but the L2 cache size is increased to 1024 KB per core. On the ARM side, there is 32 KB of L1D and 32 KB of L1P cache per core, and a coherent 4 MB L2 cache. The level 2 shared memory is increased to 6 MB and is accessible by all ARM and DSP cores. This SoC brings the ARM-based processor and DSP accelerator together in the same memory address space, along with infrastructure class I/O peripherals. It therefore provides an attractive low-power alternative for HPC applications.

3 Bare-Metal Implementation of OpenMP on C66X DSP

Most compilers translate OpenMP into multi-threaded code with calls to a custom runtime library, either via outlining [8] or inlining [9]. Because many execution details are often unknown in advance, much of the actual work of assigning computations must be performed dynamically. Part of the implementation complexity is in ensuring that the presence of OpenMP constructs does not impede sequential optimization in the compiler. An efficient runtime library to support thread management and scheduling, and shared memory and fine-grained synchronization execution is essential.

The basic hardware and operating environment of the DSP cores on the Keystone I and II systems presents some special challenges when seeking to support the OpenMP programming model. Notably the shared memory controller in Keystone devices does not maintain coherency between the C66X subsystems, and it is the responsibility of the running program to use synchronization mechanisms and cache control operations to maintain coherent views of the memories. (Coherency within a given C6X subsystem for all levels of memory is maintained by the hardware).

Traditionally, application codes executing across multiple C6X subsystems are required to explicitly manage thread synchronization and cache coherence, and communicate via the shared-L2 and shared-L3 memories. A processor can transfer a data buffer to the local-L2 via a direct memory access (DMA) controller. The hardware maintains L1D cache coherency with the local-L2 for DMA accesses. Also, the DMA transfer completion event can be used as a synchronization event between the data producer and data consumer. There is no virtual memory management unit (MMU), but a memory protection mechanism protects some shared memory from being accessed by a non-authorized processor.

TI provides a light-weight multi-core task dispatch API called Open Event Machine (OpenEM). OpenEM is designed to require minimal memory and CPU cycles [10]. OpenEM is implemented to leverage the C66X SoC's Navigator hardware queues. Various types of interactions between cores, such as blocking, communication and synchronization, are implemented by OpenEM. OpenEM also provides a fast, shared, thread-safe memory management system that is used to allocate/deallocate memory in the runtime.

An understanding of the memory model used by OpenMP is fundamental to its implementation on the Keystone I/II systems. In this respect, OpenMP specifies a *relaxed consistency* memory model that is close to weak consistency [11].

In this model threads execute in parallel with a temporary view of shared memory until they reach memory synchronization or *flush* points in the execution flow. At a flush point, threads are required to write back and invalidate their temporary view of memory. After the memory synchronization point, threads again have a temporary view of memory.

Although the C66x provides a shared memory, its consistency is not automatically maintained by the hardware. It is the responsibility of the OpenMP runtime library to perform the appropriate cache control operations to maintain the consistency of the shared memory when required.

3.1 Memory Model

OpenMP has both shared and private variables. Each thread has its own copy of a private variable that the other threads cannot access. There is only one copy of a shared variable, and all threads can access it. Private variables are located on the stack of each thread of execution. The stack can be placed in any of on-chip local, on-chip shared or off-chip shared memory.

OpenMP requires that threads synchronize their local view of shared variables with the global view at a set of implicit and explicit flush points defined in the OpenMP specification. The runtime performs this synchronization in software. The synchronization steps depend on whether the shared variable is placed in on-chip local memory (L2SRAM) or on-chip/off-chip shared memory (MSMCSRAM/DDR) as follows:

- Shared variables in on-chip “local” scratch memory(L2SRAM)
  1. L2SRAM on a core is accessible to external DSP cores via a global address space
  2. Any updates to L2 scratch by external DSP cores are kept coherent by the memory subsystem
  3. The runtime performs a write-back invalidate of L1 at all flush points.
- Shared variables in on-chip/off-chip shared memory (MSMCSRAM/DDR)
  1. Shared memory regions are marked write-through
  2. The runtime performs cache invalidate operations at all flush points. Since write-through is enabled shared memory has already been updated and there is no need to write-back data.

3.2 Parallel Regions

The essential parts of the OpenMP runtime library are implemented using the OpenEM API. For each parallel region, the OpenMP compiler divides the workload into chunks that are assigned to OpenEM tasks (micro-tasks) at runtime. One of the DSP cores is treated as a master core and the other cores are worker cores. The master core runs the main thread of execution. It is responsible for initializing the OpenMP runtime and starts executing the OpenMP program (main). The worker cores wait in a dispatch loop for OpenEM tasks to show up in a queue.

A parallel region's fork-join mechanism is implemented by the following steps:

1. After initialization, worker threads wait in a dispatch loop and check a task queue for micro-task execution notification.
2. The master thread assigns micro-tasks to worker threads by posting the micro-tasks to an OpenEM queue. The micro-task description includes a function pointer and a data pointer. It also initializes a shared counter to the number of micro-tasks generated.
3. Worker cores pull micro-tasks out of the OpenEM queue. Upon receipt of the micro-task, each worker thread executes the micro-task specified by the function pointer. The data pointer is passed as an argument to the micro-task.
4. Upon completion of a micro-task, the worker core that executed the micro-task decrements the shared micro-task counter.
5. After the master completes the execution of its own chunk, it waits for the shared micro-task counter to reach 0, indicating that all workers have completed their micro-tasks.

3.3 Synchronization

The runtime implements three methods of synchronization depending on what is being synchronized:

1. To synchronize master and worker cores during runtime initialization, a fast synchronization mechanism is implemented using coherent shared memory to store a vector. Each core independently sets or clears an element in the vector. Every core can concurrently query the entire vector by using a single 64-bit memory access. As shown in Figure 2, this mechanism is based on Lamport's Bakery algorithm [12]. The buffers are stored in non-cacheable shared memory. The message queue is used at the start of a parallel region, but all other synchronizations are performed using this new mechanism.

```
void sync(char buf0[8], char buf1[8])
{
    int core_id = get_core_id();

    buf1[core_id] = 1;
    buf0[core_id] = 0;
    /* wait until all threads have cleared buf0 */
    while (*(volatile long long *)buf0 != 0) ;

    buf1[core_id] = 0;
    /* wait until all threads have cleared buf1 */
    while (*(volatile long long *)buf1 != 0) ;

    /* reset buf0 */
    buf0[core_id] = 1;
}
```

Fig. 2. Fast synchronization mechanism using coherent shared memory

2. To synchronize master and worker cores at the end of a parallel region, a shared counter, as described in Section 3.2 is used.
3. For implicit and explicit OpenMP barriers, the sense reversing barrier shown in Figure 3 is used. This has a counter that keeps track of the number of cores participating in the barrier and a sense flag to allow the barrier to be re-used. To avoid coherency overheads, the barrier variable is placed in non-cached memory.

```
void sense_reversing_barrier(Barrier *barrier)
{
    /* To allow re-use, the barrier contains a sense variation */
    char mysense = !barrier->sense;

    if (atomic_decrement(barrier->count) == 1)
    {
        /* Last thread resets the sense and count */
        barrier->count = barrier->size;
        barrier->sense = !(barrier->sense);
    }
    else
    {
        /* Modification of sense represents end of the barrier */
        while (mysense != barrier->sense);
    }
}
```

Fig. 3. Sense reversing barrier

4 ARM to DSP OpenMP Dispatch

We have implemented an early prototype of the OpenMP 4.0 accelerator extension [13,14] *target* construct. Our prototype implementation uses the *dispatch* keyword along with memory *copyin* and *copyout*. A host program uses the dispatch construct in the following way:

```
void foo(int *in1, int *in2, int *out1, int count)
{
    #pragma omp dispatch copyin  (in1[0:count-1], in2[0:count-1], count) \
                         copyout (out1[0:count-1])
    {
        #pragma omp parallel shared(in1, in2, out1)
        {
            int i;
            #pragma omp for
            for (i = 0; i < count; i++)
                out1[i] = in1[i] + in2[i];
        }
    }
}
```

Fig. 4. Usage of dispatch construct in host program

A source-to-source translator is used to transform the initial source file with code outlined in Figure 4 to produce the ARM host side and DSP target side annotated code, as shown in Figure 5. The ARM host annotations include data movement calls. In Keystone II, these calls resort to memory maps using the UNIX *mmap* system call to leverage physical shared memory between the ARM and DSP. Therefore copyin/copyout operations have minimal overhead. The DSP source annotations made by the translator include standard OpenMP pragma additions. These two separate source files are then compiled with the gcc toolchain on the ARM side, and the TI Code Generation Tools OpenMP compiler toolchain and our OpenMP DSP runtime library on the DSP side to produce a fat executable with the DSP binary embedded inside the ARM executable. This is launched from the ARM using the TI Multi-Process Manager which loads and runs the DSP binary and the ARM host executable. Shared memory regions are set up, synchronization messages are exchanged between the ARM and DSP and the required functions are then run on DSP, which writes back the result to shared memory. An initial implementation of the dispatch construct was tested on the Appleton EVM which has a TCI6614 SoC [15] with ARM Cortex-A8 and quad-core C66X DSP on-chip. Porting the dispatch construct to utilize our current implementation of the OpenMP runtime on Keystone II is in progress.

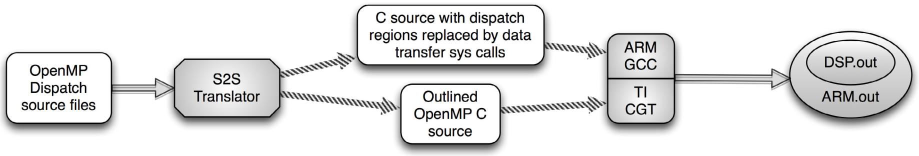

OpenMP Dispatch source files
S2S Translator
C source with dispatch regions replaced by data transfer sys calls
Outlined OpenMP C source
ARM GCC
TI CGT
DSP.out
ARM.out

Fig. 5. Dispatch executable compilation strategy

5 Evaluation Using Micro-benchmarks

In order to evaluate the performance of our DSP runtime implementation, we used the EPCC v3 benchmark [3]. For comparison we also collected data across other contemporary ARM and Intel platforms. The EPCC suite of micro-benchmarks measure the time overheads associated with invoking the different OpenMP constructs. For example, the cost of *parallel* to create a parallel region or *barrier* to synchronize threads. Here we report the overheads associated with some of the most widely used constructs.

Table 1 lists the systems considered and their main characteristics. Two different Intel platforms with at least four physical cores were considered. The Core2 Q9400 Yorkfield processor, has four cores running at 2.66 Ghz. The hexa-core Xeon X5650 Westmere processor is part of a dual-socket system and runs at 2.66 Ghz. The ARM platforms considered, include the Keystone II Hawking EVM's ARM Cortex A15 quad-core processor running at 625 Mhz and a Samsung Exynos 4412 prime SoC with quad-core ARM Cortex-A9 processors running at 2 Ghz. The ARM Cortex A15 is referred to as Hawking-A15 and the A9 as Exynos-A9. The octa-core C66X DSP processor in Keystone II Hawking EVM ran at 983 Mhz and is referred to as Hawking-DSP.

Table 1. Platforms Used in Benchmarks

PROCESSOR	**CODENAME**	**Threads/Cores/Ghz**	**Memory**
Intel Core2 Q9400	YorkField	4/4/2.67	8GB DDR3
Intel Xeon X5650	Westmere	12/6/2.67	24GB DDR3
Samsung Exynos 4412 (ARM)	Odroid-X2	4/4.Cortex-A9/2.0	2GB LPDDR2
TI Keystone II (ARM)	Hawking	4/4.Cortex-A15/0.625	2GB DDR3
TI Keystone II (DSP)	Hawking	8/8.C6678 DSP/0.983	2GB DDR3

5.1 Compilers and Tools

For the Intel Westmere platform we used GCC 4.6.4 and ICC 13.1.1 (compatible with GCC 4.6) to separately compile and run the benchmarks. These versions are denoted as X5650-GCC and X5650-ICC. They were linked against libgomp and libiomp5 respectively. On the Intel Yorkfield and ARM platforms GCC 4.7.3 with libgomp was used. The compiler option *-mcpu=cortex-a9* was used on the Exynos and *-mcpu=cortex-a15* on the Hawking. In addition both ARM platforms used the *-mfpu=neon,mfloat-abi=hard* compiler flag. TI Code Generation

Tools 7.4.2, XDC Tools 3.24.05.48, OpenEM 1.2.0.1, PDK Keystone2 1.00.00.09, PDK C6678 1.1.2.6 along with our current version of the OpenMP runtime were used to create the executable for the C6678 DSP. All platforms except the Hawking-A15 and the DSPs were running Ubuntu Linux with kernel version greater than 3.0. The Hawking ARM cores used a custom distribution of Linux, called Arago, built specifically for the Hawking EVM. It includes the 3.8.4 Linux kernel. On the Linux hosts, the `OMP_PROC_BIND` and `GOMP_CPU_AFFINITY` environment variables were set to bind threads to processor cores and to prevent thread migration between cores. For timing measurements on the Intel and ARM platforms, the EPCC v3 timer function `getclock()` which uses `omp_get_wtime()` remained unchanged and reported times in microseconds. Measurements on the DSP required modifications to the timer function. A native time-stamp counter was used to measure the exact CPU cycles elapsed as shown in Figure 6. For direct comparison of all platforms, all time measurements were normalized w.r.t CPU clock speed and reported in CPU cycles using the equation, $cpu\_cycles = overhead\_time(\mu s) * mhz$.

```
/* Wall cycles using TSC_read */
void wcycles(unsigned long long *c)
{
    static int first = 1;
    extern void TSC_enable(void);
    extern unsigned long long TSC_read(void);
    if (first)
    {
        TSC_enable();
        first = 0;
    }
    *c = TSC_read();
}

/* TSC_enable Assembly Code */
.global TSC_enable

TSC_enable:

RETNOP  B3, 4
MVC     B4, TSCL   ; writing any value enables timer

/* TSC_read Assembly Code*/
.global TSC_read

TSC_read:

RETNOP  B3, 2
DINT
MVC TSCH,   B5    ; Read the snapshot of the high half
MV  B5,  A5
```

Fig. 6. Measuring cpu cycles on the DSP

5.2 Discussion

A crucial difference between the multi-core DSP in the Keystone architectures and other processors evaluated in this study is cache coherence. While the Intel and ARM multi-core processors have hardware managed cache coherence protocols, programs running on the multi-core DSP have to ensure cache coherence in software. As explained in Section 3, the Keystone shared memory controller does not ensure this memory consistency. As a result we perform *flush* operations at implicit and explicit synchronization points in our OpenMP runtime, which invalidate L1 and L2 caches and write them back to main memory. In our

Table 2. Cost of software managed cache coherency operation for DSP (cycles)

DSP (L2 = 0)	1 Thread	2 Threads	4 Threads	6 Threads	8 Threads
Hawking-DSP	1350	1355	1357	1353	1364

evaluation, we set DSP L2 cache to be 0K to minimize flush overhead. Table 2 presents operation cycle counts on Hawking DSPs averaged over 200 iterations. This shows the cost to be roughly 1350 cycles regardless of thread count.

Results for the EPCC benchmarks are given in Figure 7. In each bar-graph the cpu cycle overhead values are given for each platform using 1-8 OpenMP threads. For platforms with 4 cores, only results with up to 4 threads are given.The PARALLEL construct is most fundamental, specifying the creation of an OpenMP parallel region and spawning a team of threads. Each of the platforms in Figure 7(a) demonstrates the expected behaviour of increasing cycles overhead with increasing number of threads. The X5650-ICC results show the least 1-thread overhead time of 585 cycles. Not surprisingly a sharp increase is seen on the X5650 with both the GCC and ICC compilers in going from 6-thread to 8-threads as the last 2 threads are spawned on a different socket. The Exynos and Hawking ARM processors show comparable overheads to the Intel platforms. Overheads for more than 1-threads observed on the DSP are higher than Intel and ARM platforms. However, each parallel region incorporates implicit cache-coherence flush operations. Discounting the cost of these flush operations, performance of the OpenMP DSP runtime is comparable to Intel and ARM processors.

The BARRIER construct is used to specify an explicit synchronization point inside a parallel region which all threads must reach for any to progress beyond that point. As shown in Figure 7(b), X5650-ICC performs the best among all platforms across all thread configurations. Similar to PARALLEL overheads, the ARM platforms have comparable times to Intel. The Hawking DSP has cycle overheads of between 1800 and 3206. Subtracting the cost of 1 flush operation from these yields overheads of between 450 and 1842. The latter are almost on a par with the X5650-GCC values. The FOR construct is used to split for loops between thread and data in Figure 7(c) and show similar patterns to those observed for PARALLEL and BARRIER.

The STATIC constructs are used to specify compile-time scheduling of loop iterations between threads. STATIC 1 indicates that each thread gets 1 loop iteration to process at a time, while STATIC 128 gives 128 loop iterations at a time. Figure 7(d) and 7(e) show that the DSP platforms perform significantly better than the Intel platforms, while the ARM platforms perform the best overall. This suggests that the chunk sizes of 1 and 128 are not ideal for the memory hierarchy and architecture of the Intel platforms, but are more suited for the ARM and DSP platforms. It also shows that the DSP OpenMP runtime performs effective static scheduling for these chunk sizes. The DYNAMIC construct is similar to STATIC in that it partitions the scheduling of loop iterations between threads. However, in contrast to STATIC the loop iterations are now partitioned dynamically with the next available thread executing the next loop iteration.

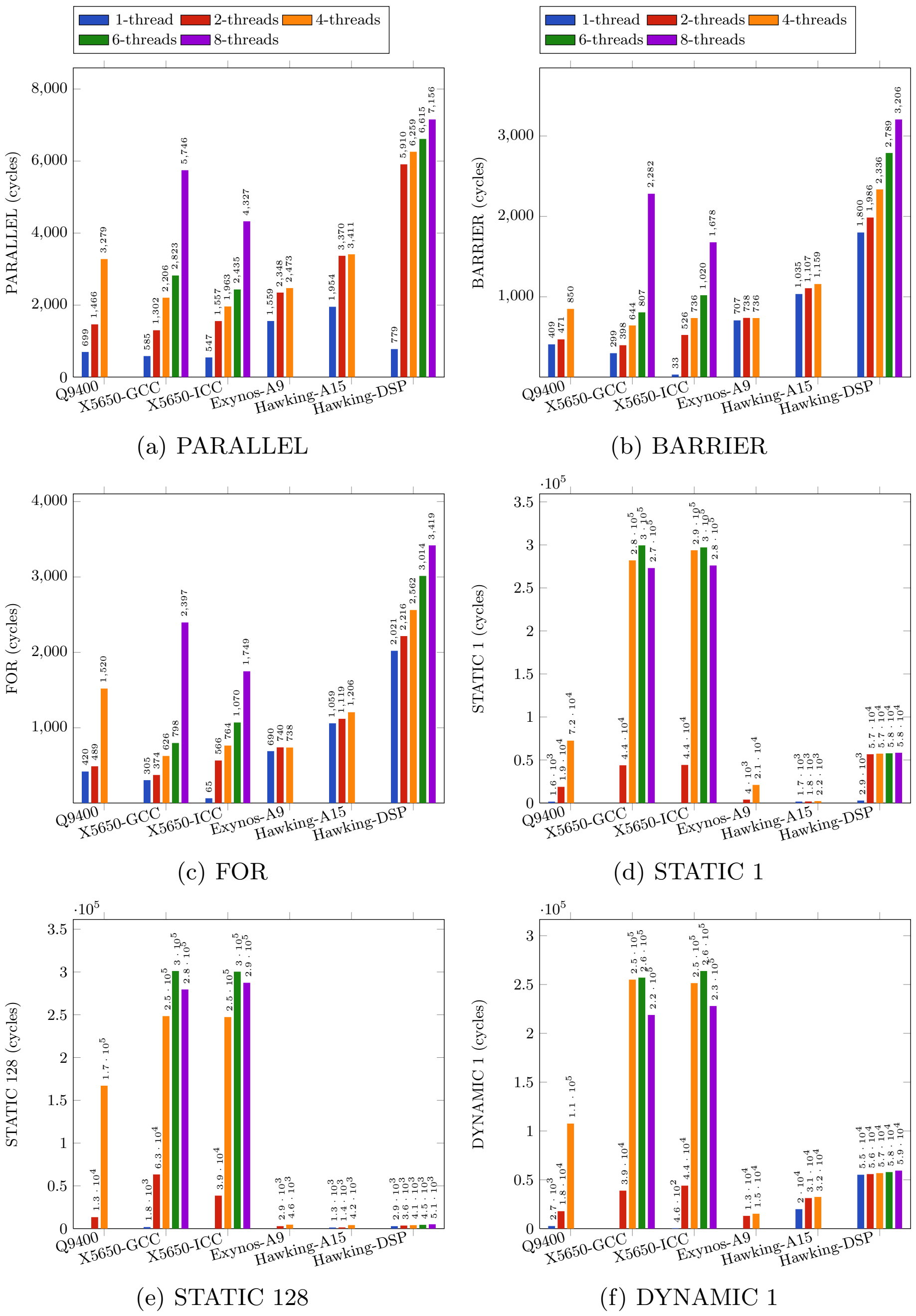


Fig. 7. Cost comparison of OpenMP constructs in CPU cycles

This permits load balancing when iterations give rise to variable work. Interestingly the results for DYNAMIC 1 scheduling in Figure 7(f) are very similar to those for STATIC 1.

6 Related Work

Use of TI C66X DSPs for HPC is demonstrated in [2,16,17,18]. Directive based programming for GPU accelerators has most recently been standardized by OpenACC [19] after previous implementations such as hiCUDA [20] and PGI Accelerator Model [21]. IBM provides an OpenMP compiler [22] and runtime library for the Cell Broadband Engine. Extensions to OpenMP to support accelerators were introduced in [23,24,25,14]. Various RTOSs such as SYS/BIOS [26] and DSP/BIOS [10] have been used on the C6678 DSP. [27] provides an OpenMP runtime using DSP/BIOS for the TI C64x DSP. A bare-metal implementation of an OpenMP runtime for the Cradle CT3400 multi-core DSP is outlined in [28].

7 Conclusions and Future Work

We have presented our initial experiences with implementing a bare-metal OpenMP runtime for the Keystone II C66X multi-core DSP. We addressed various challenges such as lack of memory management units and cache coherence as part of our implementation process. CPU cycle overheads for various OpenMP synchronization and scheduling constructs were measured using the EPCC micro-benchmarks on the C66X DSP and other contemporary Intel and ARM processors. The results demonstrated that our DSP runtime performed at par or better than Intel and GCC OpenMP runtimes for most OpenMP constructs. Software managed cache coherence was acknowledged to be a limiting factor for DSP runtime performance. Porting and evaluating the efficiency and performance of the OpenMP accelerator dispatch construct to Keystone II is a future work item. Evaluation of OpenMP task performance on C66X DSP and implementation of task dispatch from ARM to DSP is of interest. Measurement of energy efficiency and performance of various application codes using OpenMP is also of interest.

Acknowledgements. This work is supported in part by the Australian Research Council Discovery Project DP0987773. We thank Andrew Tridgell for providing us with an the ODROID-X2 development platform.

References

1. Mitra, G., Johnston, B., Rendell, A.P., McCreath, E., Zhou, J.: Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW). IEEE (2013)

2. Igual, F.D., Ali, M., Friedmann, A., Stotzer, E., Wentz, T., van de Geijn, R.A.: Unleashing the high-performance and low-power of multi-core dsps for general-purpose hpc. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, vol. 26. IEEE Computer Society Press (2012)
3. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for openMP tasks. In: Chapman, B.M., et al. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 271–274. Springer, Heidelberg (2012)
4. Texas Instruments Literature: SPRUGH7: TMS320C66x DSP CPU and Instruction Set Reference Guide
5. Texas Instruments Literature: SPRS691C: TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor
6. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco (2003)
7. Texas Instruments Literature: SPRS866: 66AK2H12/06 Multicore DSP+ARM Keystone II System-on-Chip (SoC)
8. Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience 12(12), 1193–1203 (2000)
9. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An optimizing, portable OpenMP compiler. In: Concurrency and Computation: Practice and Experience, Special Issueon CPC 2006 selected papers (2006) (accepted)
10. Texas Instruments Literature: SPRU423D: DSP/BIOS user's guide
11. Hoeflinger, J.P., de Supinski, B.R.: The openmp memory model. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005/2006. LNCS, vol. 4315, pp. 167–177. Springer, Heidelberg (2008)
12. Lamport, L.: The parallel execution of do loops. Commun. ACM 17(2), 83–93 (1974)
13. OpenMP, A.: Openmp application program interface, v. 4.0 - rc 2 (2013)
14. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011)
15. Texas Instruments Literature: SPRT610: TMS320TCI6612/14 High Performance comes to small cell base stations
16. Ali, M., Stotzer, E., Igual, F.D., van de Geijn, R.A.: Level-3 blas on the ti c6678 multi-core dsp. In: 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 179–186. IEEE (2012)
17. Ahmad, A., Ali, M., South, F., Monroy, G.L., Adie, S.G., Shemonski, N., Carney, P.S., Boppart, S.A.: Interferometric synthetic aperture microscopy implementation on a floating point multi-core digital signal processer. In: SPIE BiOS, International Society for Optics and Photonics, p. 857134 (2013)
18. Note, F.W., Van Zee, F.G., Smith, T., Igual, F.D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T.M., et al.: Implementing level-3 blas with blis: Early experience (2013)
19. Reyes, R., Lopez, I., Fumero, J.J., de Sande, F.: Directive-based programming for gpus: A comparative study. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 410–417. IEEE (2012)

20. Han, T.D., Abdelrahman, T.S.: hi cuda: a high-level directive-based language for gpu programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 52–61. ACM (2009)
21. Wolfe, M.: Implementing the pgi accelerator model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 43–50. ACM (2010)
22. Eichenberger, A.E., O'Brien, J.K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., et al.: Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. IBM Systems Journal 45(1), 59–84 (2006)
23. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A proposal to extend the openMP tasking model for heterogeneous architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009)
24. Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguade, E., Jiménez-González, D.: Openmp extensions for fpga accelerators. In: International Symposium on Systems, Architectures, Modeling, and Simulation, SAMOS 2009, pp. 17–24. IEEE (2009)
25. Ayguadé, E., Badia, R.M., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzàlez, M., Igual, F., Jiménez-González, D., Labarta, J., et al.: Extending openmp to survive the heterogeneous multi-core era. International Journal of Parallel Programming 38(5-6), 440–459 (2010)
26. Texas Instruments Literature: SPRUGO6A: SYS/BIOS inter-processor communication (IPC) and I/O user's guide
27. Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., Gatherer, A.: Implementing openmp on a high performance embedded multicore mpsoc. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–8. IEEE (2009)
28. Jeun, W.C., Ha, S.: Effective openmp implementation and translation for multiprocessor system-on-chip without using os. In: Proceedings of the 2007 Asia and South Pacific Design Automation Conference, pp. 44–49. IEEE Computer Society (2007)