HetroOMP: OpenMP for Hybrid Load Balancing Across Heterogeneous Processors

Vivek Kumar[1](✉), Abhiprayah Tiwari[1], and Gaurav Mitra[2]

[1] IIIT-Delhi, New Delhi, India
vivekk@iiitd.ac.in
[2] Texas Instruments, Houston, USA

Abstract. The OpenMP accelerator model enables an efficient method of offloading computation from host CPU cores to accelerator devices. However, it leaves it up to the programmer to try and utilize CPU cores while offloading computation to an accelerator. In this paper, we propose *HetroOMP*, an extension of the OpenMP accelerator model that supports a new clause `hetro` which enables computation to execute simultaneously across both host and accelerator devices using standard tasking and work-sharing pragmas.

To illustrate our proposal for a hybrid execution model, we implemented a proof-of-concept work-stealing HetroOMP runtime for the heterogeneous TI Keystone-II MPSoC. This MPSoC has host ARM CPU cores alongside accelerator Digital Signal Processor (DSP) cores. We present the design and implementation of the HetroOMP runtime and use several well-known benchmarks to demonstrate that HetroOMP achieves a geometric mean speedup of 3.6× compared to merely using the OpenMP accelerator model.

Keywords: OpenMP accelerator model · Heterogeneous architectures · Hybrid work-stealing

1 Introduction

Modern processor design relies heavily on heterogeneity to deliver high performance and energy-efficiency. As a result, contemporary High Performance Computing (HPC) systems are widely composed of accelerator devices alongside multi-core CPU processors. Popular accelerator devices include Graphics Processing Units [21] (GPU) and Field Programmable Gate Arrays [25] (FPGA), while more unconventional accelerators include Digital Signal Processors [17] (DSP). Such accelerators can be targeted using popular programming models such as Nvidia's CUDA [20] and Khronos OpenCL [19]. However, understanding how to use CUDA and OpenCL efficiently is non-trivial. The OpenMP 4.0 [2] accelerator model was introduced to address this issue. It provides a high-level, portable and compiler directive-based interface which aims to have a much smaller learning curve compared to both CUDA and OpenCL. The accelerator model is host-centric where the programmer designates regions of code to be offloaded from the host to an accelerator device while orchestrating a map/copy of input and output data for that region as required. The OpenMP compiler then generates accelerator specific low-level code and API calls into an OpenMP runtime environment which manages input data transfers between host and accelerator, launches compute kernels on the accelerator and transfers results back from the accelerator to the host. Although this approach enables high programmer productivity, a major limitation is that it does not target both host and accelerator devices simultaneously. While offloading code to an accelerator, the onus is on the programmer to manually partition the workload and run a computation on the host CPU cores.

Several factors affect the efficiency of manual partitioning: (i) the host and accelerator devices might have very different performance characteristics; (ii) there may be high communication latency between host and accelerator affecting partition granularity; and (iii) there may be several layers of parallelism in a compute kernel. These factors make manual partitioning an NP-hard problem. In most cases host CPU cores remain idle or busy-wait for accelerator cores to finish computation, thereby wasting CPU cycles and reducing energy-efficiency. OpenMP does not provide default support to best utilize both host and accelerator resources on a system. In this paper we target this limitation of the OpenMP accelerator model by focusing on two research questions:

RQ1: Without affecting programmer productivity, is it possible to extend the OpenMP accelerator model to identify computation suitable for hybrid execution over both host and accelerator device?

RQ2: Is it possible to design and implement a high-performance OpenMP runtime that could *dynamically* load balance computation across heterogeneous processing elements?

To address `RQ1`, we propose HetroOMP, an extension of the OpenMP accelerator model with a new clause "`hetro`". It enables execution of OpenMP `task` and parallel `for` loops simultaneously across both host and accelerator devices using compiler source-to-source translation. The critical focus on energy-efficiency has led the HPC community to consider low-power heterogeneous ARM SoC based embedded systems with various accelerators (GPU, DSP) on-chip as possible alternatives to conventional HPC systems. To address `RQ2` we use such an embedded system, the Texas Instruments Keystone II [24] *Hawking* (K2H) Multi-Processor System-on-chip (MPSoC) which houses a quad-core ARM CPU and eight-core DSP accelerator on-chip.

We present the design of a novel, lightweight work-stealing [6] runtime implemented on K2H which enables high-performance load-balancing across both ARM and DSP cores. Several OpenMP tasking and `parallel for` benchmarks are used to compare the performance of the HetroOMP runtime to the default OpenMP device and host-only executions using the TI OpenMP runtime. We show that HetroOMP is highly competitive and can outperform default OpenMP. In summary, this paper makes the following contributions:

- HetroOMP, an extension to OpenMP accelerator model, which enables hybrid parallelism across host and accelerator device.
- A lightweight runtime implementation of HetroOMP that uses work-stealing for dynamic load-balancing across heterogeneous processing elements.
- Evaluation of HetroOMP on TI Keystone-II MPSoC by using several well-known tasking and `parallel for` benchmarks.

2 Related Work

OpenACC [26] is a directive-based programming model for Nvidia GPUs. OmpSs [10] extended the OpenMP task directives to the StarSs [22] programming model supporting kernel offloads for GPUs and FPGAs [10]. Chapman et al. [7] and Mitra et al. [18] presented implementations of OpenMP accelerator model for TI Keystone-II MPSoC. Mitra et al. further improved the OpenMP implementation for TI Keystone-II MPSoC [4] by presenting a framework that automatically addressed the parallelization of code annotated with OpenMP 4.0 directives. However, none of these implementations support dynamic load balancing across both host and accelerators.

There have been prior studies on hybrid execution across host and device. Luk et al. [16] presented a heterogeneous programming model that automatically partitioned loop level parallelism across host and GPU for hybrid execution. Barik et al. [5] presented another such hybrid programming model for CPU-GPU platforms. A common limitation in both these studies is that a prior training run of an application is mandatory to discover the optimal work partition ratio. Ozen et al. proposed extensions to OpenMP accelerator model to support hybrid execution across CPU and GPU. As GPUs are mostly suited for data-parallelism, their proposed extensions were tailored for work-sharing pragmas. Linderman et al. proposed a map-reduce based programming model for automatic distribution of computations across heterogeneous cores and evaluated it over a CPU-GPU based processor [15]. CnC-HC programming model [23] provided a work-stealing based dynamic load balancing across CPUs, GPUs and FPGAs. Kumar et al. [13] presented HC-K2H programming model for TI Keystone-II MPSoC that used a hybrid work-stealing runtime for dynamic load balancing across ARM and DSP cores (Sect. 3.2).

3 Background

3.1 TI Keystone-II MPSoC

Recent work [17] has considered the TI ARM/DSP K2H SoC for HPC workloads. It has 4 ARM Cortex-A15 cores running at up to 1.4 GHz and 8 TI C66x floating-point DSP cores running at up to 1.2 GHz. The ARM cores have 32 KB of L1 cache each and 4 MB of shared L2 cache while DSP cores can have 32 KB of L1 cache and 1 MB of L2 cache each. The ARM cores have hardware managed cache coherence, while the DSP cores do not. Additionally, there is no cache coherence between ARM and DSP cores. Both ARM and DSP cores share the same memory bus to off-chip DDR memory but have separate address spaces. The Multicore Shared Memory Controller in K2H provides 6 MB of shared scratchpad memory (SRAM) between ARM and DSP cores. The Multicore Navigator provides hardware queues (henceforth mentioned as *HardwareQueue*) that can be used to communicate and dispatch tasks between ARM and DSP cores. There are two queue managers with 8192 queues each and 64 descriptor memory regions per queue manager.

3.2 Hybrid Work-Stealing Methodology

We address `RQ2` (Sect. 1) by implementing a hybrid work-stealing runtime. It shares characteristics with HC-K2H [13] which supported an `async–finish` [8] based parallel programming model. An `async–finish` program is represented as "`finish{ async S1; S2 } S3;`". Here, the `async` clause creates a task `S1` that could run in parallel to task `S2`. Statement `finish` starts a finish scope and ensures both tasks `S1` and `S2` are completed before starting the execution of `S3`. HC-K2H supports the `forasync` loop-level parallelism construct which recursively divides a `for` loop's iterations into two halves with each recursion step being an `async`.

HC-K2H used a hybrid work-stealing runtime for dynamic load balancing of `async` tasks across ARM and DSP cores. Work-stealing is a very efficient strategy for distributing work in a parallel system and is implemented as shown in Fig. 1. It consists of a pool of threads, where each thread (*worker*) maintains a data structure (*deque*) to *push* the local set of *tasks* (from the *tail* end). When a worker becomes idle, it attempts to *pop* a task from the tail of its deque. If it fails to pop, then it becomes a *thief* and searches for a *victim* in the thread pool from which to *steal* a task (from the *head* end).

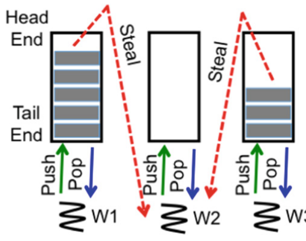


Fig. 1. Work-stealing implementation

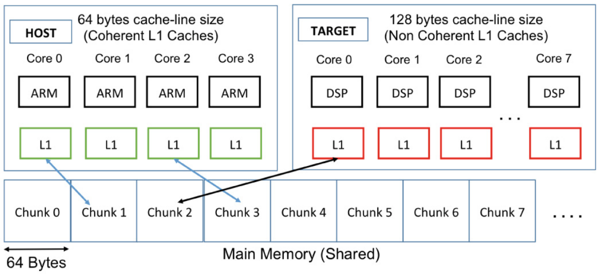
This double ended software deque (henceforth mentioned as *CilkDeque*) was introduced by the Cilk language [11]. HC-K2H used a similar CilkDeque based work-stealing implementation for ARM cores (`ARM_WS`). As DSP cores do not support CilkDeque which could be accessed directly by any other DSP or ARM cores, HC-K2H used a separate work-stealing runtime for DSP cores (`DSP_WS`) that used HardwareQueue instead of CilkDeque. A HardwareQueue differs significantly from a CilkDeque as it is not double-ended and can be used only in two modes, either as a Last-In-First-Out (LIFO) queue or as a First-In-First-Out (FIFO) queue. `DSP_WS` uses the LIFO implementation of HardwareQueue where all three operations push, pop and steal happen only from the tail end. Whenever ARM or DSP workers go idle in HC-K2H, they first attempted an *intra-arch* steal before attempting an *inter-arch* steal. ARM workers can directly perform inter-arch steals from DSP worker's HardwareQueue. As DSP workers cannot

```
int *A /*size=N*/, *B /*size=N*/, *C /*size=N*/, N;
int cache_line = omp_cache_granularity();
int MIN_CHUNK = cache_line/sizeof(int);
main() {
 int i;
 #pragma omp target map(tofrom:C[0:N]) \
     map(to:A[0:N], B[0:N], N)
 #pragma omp parallel for firstprivate(A, B, C) \
         private(i) schedule(hetro, MIN_CHUNK)
 for(i=0; i<N; i++) {
   C[i] = A[i] + B[i];
 }
}
```

(a) Parallel vector addition in HetroOMP by using work-sharing pragma

```
int* A /*size=N*/, N;
int cache_line = omp_cache_granularity();
int MIN_CHUNK = cache_line/sizeof(int);
void msort(int left, int right) {
 if(right-left > MIN_CHUNK) {
   int mid = left+(right-left)/2;
   #pragma omp task untied \
     firstprivate(left, mid) hetro(A:N)
   msort(left, mid);
   msort(mid+1, right);
   #pragma omp taskwait
   merge(left, mid, right);
 } else {
   sequentialSort(left, right);
 }
}
main() {
  #pragma omp target map(to:N) \
         map(tofrom:A[0:N])
  #pragma omp parallel \
  firstprivate(A,N) hetro
  #pragma omp single
  msort(0, N-1);
}
```

(b) Parallel recursive MergeSort in HetroOMP by using tasking pragma



(c) Example of false sharing happening across ARM and DSP cores during a hybrid execution. ARM cache line size is 64 bytes whereas for DSP it is 128 bytes. Cache write back from DSP can overwrite the results calculated by ARM unless ARM also operates on 128 bytes cache line granularity.

Fig. 2. HetroOMP programming model. Underlined code in Figs. 2(a) and (b) are HetroOMP specific code in standard OpenMP.

access ARM's CilkDeque, a shared HardwareQueue was used by ARM workers to offer tasks to DSP workers for inter-arch stealing.

4 HetroOMP Programming Model

HetroOMP addresses `RQ1` (Sect. 1) by extending the OpenMP accelerator model with a new clause, `hetro`, which can be used to perform hybrid execution of computation kernels with work-sharing and tasking pragmas. Figure 2 shows usage of the `hetro` clause in two different OpenMP programs, a `parallel for` based vector addition in Fig. 2(a), and `task`-based parallel divide-and-conquer implementation of `MergeSort` in Fig. 2(b). HetroOMP specific code in both these examples has been underlined. Removing the HetroOMP code will leave a valid OpenMP 4.0 program that simply offloads computation to the accelerator. Clause `hetro` could be used in three different ways: (a) as a clause to `pragma omp parallel` indicating the scope of hybrid execution, (b) as a parameter to `schedule` clause in `pragma omp for`, with an optional chunk size showing hybrid execution of loop iterations, and (c) as a clause to `pragma omp task` along with the name and count of all writable type shared variables in this task, e.g., "`Var1:Count1, Var2:Count2, ..., VarN:CountN`" (detailed explanation in Sect. 6.2).

False sharing is a well-known performance bottleneck in shared memory parallel programs. However, it can also affect the correctness of a HetroOMP program. This could happen due to differences in cache line sizes and cache coherency protocols across host and accelerator. To understand this, consider Fig. 2(c) that represents the execution of a HetroOMP program shown in Fig. 2(a) with a chunk size of 16 instead of `MIN_CHUNK`. The total number of chunks (tasks) generated would be N/16 with each chunk 64 bytes in size. ARM cores (host) on K2H are cache coherent with L1 cache line size of 64 bytes, whereas DSP cores (device) are not cache coherent and have L1 cache line size of 128 bytes. Cache coherent ARM cores compute Chunk1 and Chunk3 with results automatically written back to the main memory. Chunk2 is calculated later by a DSP core, and explicit write-back of L1 cache is performed for the result to appear on main memory. However, this 128 byte write-back could possibly corrupt the result of either Chunk1 or Chunk3. HetroOMP programmers can resolve this either by using chunk size in multiples of 32 (128 bytes) or by padding the result `C` vector such that each chunk is of size 128 bytes (or it's multiple). HetroOMP provides a new API `omp_get_min_ganularity` to calculate the cache line granularity. Programmers can follow conventional task cutoff techniques for controlling the task granularity of compute-bound programs that does not depend on `MIN_CHUNK` (e.g., `Fib` in Sect. 7).

5 Design of HetroOMP Runtime

5.1 Limitations of HardwareQueue

Recall from Sect. 3.2, HC-K2H uses HardwareQueue for implementing `DSP_WS` where all three operations, push, pop and steal happen at the tail end (LIFO). In spite of design simplicity, such a HardwareQueue based work-stealing runtime suffers from two subtle issues unlike the CilkDeque: (a) load imbalance among DSP cores leading to frequent steals, and (b) cache write-back and invalidation operation (henceforth mentioned as *CacheWBInv*) at *every* end finish scope.

CilkDeque is designed to support push and pop operations from the tail end (LIFO) and stealing operations from the head end (FIFO). LIFO accesses by victims improves locality whereas FIFO accesses by thieves reduce load imbalance (frequent steals). In regular divide-and-conquer applications, older tasks (available on the head) are more computationally intensive than the recently created tasks (available on tail). Hence, stealing from FIFO end will execute a more significant chunk of computation than from LIFO end. However, HardwareQueue based `DSP_WS` in HC-K2H lacked this benefit as both victim and thief workers operated from the same side, thereby leading to frequent steals.

The other limitation of HardwareQueue based `DSP_WS` is mandatory CacheWBInv at every end finish scope. Recall, HardwareQueue is directly accessible to all ARM and DSP cores. Due to this some of the `async` tasks generated within a `finish` scope can execute at cache coherent ARM cores while some of them could execute at cache incoherent DSP cores. As task owner (DSP) itself is cache incoherent, they cannot discover that an `async` task was stolen unless they perform explicit CacheWBInv at every end `finish` scope. This is a costly operation that won't affect the performance of flat `finish` based kernels (e.g., `parallel for`) but can significantly hamper the performance of task parallelism based applications containing nested `finish` scopes.

5.2 Private Deque Based `DSP_WS`

HetroOMP has been designed considering two main factors (a) CacheWBInv is not required at every end `finish` scope but it should be done only when a steal happens between two cache-incoherent processors under a finish scope, and (b) `DSP_WS` can reap the benefits of work-stealing only if it also allows steal operations from its head end (FIFO) and push/pop from its tail end (LIFO). These two factors are accounted in HetroOMP by using a *private deque* [3] (henceforth mentioned as *PvtDeque*) based implementation of `DSP_WS` instead of HardwareQueue. Acar et. al. originally introduced PvtDeque but in the context of reducing the overheads associated with memory fence operations in CilkDeque. PvtDeque differs from CilkDeque only in terms of steal operation as it doesn't allow a thief to steal a task directly. The thief has to make an entry in the *communication cell* hosted by a victim which keeps checking this communication cell during its push and pop operations. If they notice a waiting thief, they steal a task (on the thief's behalf) from the head end (FIFO) of its PvtDeque and then transfer it to the thief. For implementing a PvtDeque for each DSP core in HetroOMP, we reconfigured the default 1 MB L2 cache available to each DSP such that 512 KB remained as L2 cache and the rest as un-cached SRAM containing the PvtDeque.

6 Implementation of HetroOMP Runtime

6.1 Source-to-Source Translation of a HetroOMP Program

We extended the OpenMP-to-X [12] framework, such that it can perform source-to-source translation of HetroOMP code into a C program with calls to the HetroOMP runtime. OpenMP-to-X uses Clang LibTooling [1] and was designed to perform source-to-source translation of an OpenMP program into a HClib [14] program. Figure 3 shows the result of this source-to-source translation for the program shown in Fig. 2(b). The underlined code in Fig. 3 demonstrates the modifications to default HC-K2H program to support the HetroOMP runtime API calls. Translation of HetroOMP to C code begins from the `main` method. Pragma `target map` (Fig. 2(b), Line 18) gets replaced with an API call for variable initialization at DSP (Fig. 3, Line 43). The `tofrom` and `to` clauses are ignored as

```
1  int* A /*size=N*/, N;
2  int cache_line = DSP_CACHE_LINE; //128 bytes
3  int MIN_CHUNK = cache_line/sizeof(int);
4  void msort(int left, int right) {
5   if(right-left > MIN_CHUNK) {
6    int mid = left+(right-left)/2;
7    /*start new nested finish scope*/
8     finish=allocate();
9     setup_current_finish(finish);
10    finish->writable_vars(A, sizeof(int)*N);
11    finish->incoherentCoreSteals=false;
12   /*launch task*/
13    task=create_task(msort, left, mid);
14    if(ARM) {
15     push_CilkDeque(task);
16    }
17    if(DSP) {
18     push_PrivateL2Deque(task);
19     help_incoherentCore_steal();
20    }
21    ATOMIC(finish->pendingAsyncs++);
22   /*this will create nested async-finish*/
23   msort(mid+1, right);
24   /*end current finish scope*/
25    while(finish->pendingAsyncs>0) {
26     if(tasks_on_my_deque()>0) {
27      help_incoherentCore_steal();
28      pop_and_execute();
29     }
30     else steal_and_execute();
31    }
32    if(finish->incoherentCoreSteals) {
33     cacheWbInv(finish->get_writable_vars());
34    }
35    setup_current_finish(finish->parent);
36   /*continue seqential execution*/
37   merge(left, mid, right);
38  } else {
39   sequentialSort(left, right);
40  }
41 }
42 main() {
43  initialize_at_DSP_device(A, N);
44  hybrid_execution(true);
45  msort(0, N-1);
46 }
```

Fig. 3. Source-to-source translation of HetroOMP program shown in Fig. 2(b). All underlined code are the changes in HC-K2H runtime code to support HetroOMP.

data does not need to be copied between host and accelerator device as shared DDR memory between ARM and DSP is being utilized. Clause `hetro` on pragma `parallel` (Fig. 2(b), Line 12) indicates hybrid execution across host and accelerator (Fig. 3, Line 44). Without the `hetro` clause DSP-only offload will occur. The translation of code in Fig. 2(a) happens in a similar fashion. The only difference being, the pragma `parallel for` will be converted to a `forasync` API with chunk size `MIN_CHUNK` to avoid false sharing.

In this prototype implementation of the HetroOMP translator, a naive approach toward source code translation for pragma `task` and pragma `taskwait` is adopted. The pragma `task` is replaced with an `async` creation (Fig. 3, Lines 13–21) and pragma `taskwait` is replaced with an end finish scope (Fig. 3, Lines 25–35). In order to decide when to generate the start finish scope, a boolean flag is used. It is set to true at pragma `parallel` (Fig. 2(b), Line 20). After this when the translator encounters a pragma `task` (Fig. 2(b), Line 7), it will first generate a start finish scope (Fig. 3, Lines 8–11) followed by an `async` creation. The boolean flag is then set to false. Any further pragma `task` will then be translated to an `async` only. This flag is reset to true again at pragma `taskwait` (Fig. 2(b), Line 11).

6.2 HetroOMP Code Flow

In Fig. 3, the call to recursive `msort` method first creates a `finish` object at Line 8. This `finish` object then stores the pointer to its parent `finish` (Line 9), and the list of writable type shared variables under this `finish` scope (Line 10). These writable type shared variables are the ones indicated by the user in the `hetro` clause to `pragma omp task` (Fig. 2(b), Line 8).

HetroOMP has a boolean counter `incoherentCoreSeals` at each finish (Fig. 3, Line 11) for tracking when CacheWBInv is required at the end finish scope. ARM cores in HetroOMP directly push a task to CilkDeque (Line 15). HC-K2H did it differently as in that case ARM cores pushed few tasks to shared HardwareQueue in advance for DSPs to steal. Delaying it until an actual steal request from DSP helps HetroOMP understand the `finish` scope from where a task went to a cache incoherent core. A DSP core in HetroOMP first pushes a task to the tail of its PvtDeque (Line 18) and then executes the method `help_incoherentCore_steal` to transfer a task to any waiting thief (Line 19). At the end finish scope both ARM and DSP cores find and execute tasks until there are no more pending under this `finish` scope (Line 25). If tasks are available on local deque then while inspecting, both ARM and DSP workers in HetroOMP first execute the method `help_incoherentCore_steal` (Line 27) for transferring a task to any waiting thief followed by popping a task for self-execution (Line 28). Each DSP in HetroOMP has a dedicated HardwareQueue based communication cell where a thief (ARM/DSP) can indicate its steal request. Another Hardware-Queue is shared between ARM and DSP where an ARM core can push a task for DSPs to steal. For transferring a task to a waiting thief (at an incoherent core), a DSP can steal a task from the head of its PvtDeque and then move it to waiting thief (ARM/DSP), whereas an ARM worker steals a task from the head of its CilkDeque and pushes it to shared HardwareQueue (for DSP). As a thief receives a task from the head end of either of the deques, the number of steals can be reduced between cache incoherent workers. Whenever a core transfers a task via `help_incoherentCore_steal`, it will first perform a CacheWBInv followed by updating the counter `incoherentCoreSeals` in the current `finish` as `true`. For stealing, each core first attempts an intra-arch steal followed by an inter-arch steal upon failing (Line 30). Once out of the spin loop but before ending current `finish` scope, both ARM and DSP will do a CacheWBInv for all writable type shared variables (Line 33) based on the status of `incoherentCoreSeals` (Line 32). This technique avoids costly cache flushes in HetroOMP at every end `finish` scope.

7 Experimental Methodology

Across all experimental evaluations two broad categories of OpenMP benchmarks were used: (a) recursive divide-and-conquer applications that used nested `task` and `taskwait` pragmas, and (b) applications using parallel for loop pragmas. Each of these benchmarks is described in Table 1. We have chosen only those benchmarks where it was straightforward to remove false sharing between ARM and DSP either by loop tiling, by padding of shared data structures, or by altering task granularity. Padding was only applied as last strategy.

Five different versions of each benchmark were used: (a) OpenMP `ARM-only` implementation that runs only on ARM cores, (b) OpenMP `DSP-only` implementation that runs only on DSP cores, (c) HetroOMP version that uses the `hetro` clause but supports all 3 configurations (`ARM-only`, `DSP-only`, and `Hybrid`), (d) HC-K2H version that also supports all 3 configurations similar to HetroOMP, and (e) Sequential ARM implementation that executes on ARM and is obtained

Table 1. Benchmarks used for the evaluation of HetroOMP

Name	Description	Common settings	Source	OpenMP category
`Fib`	Calculate Nth Fibonacci number	N = 40. Task cutoff at N = 20	HC-K2H [13]	Tasking
`Matmul`	Multiplication of two matrices	Size = 1024 × 1024. Task cutoff at 6xMIN_CHUNK	Cilk [11]	Tasking
`Knapsack`	Solves 0–1 knapsack problem using branch and bound technique	N = 500 and capacity = 20000. Task cutoff at depth = 10	Cilk	Tasking
`MergeSort`	Merge sort algorithm	Array size = 4096 × 4096. Task cutoff at 4xMIN_CHUNK	Authors	Tasking
`Heat`	Heat diffusion using Jacobi type iterations	nx = 8192, ny = 2048 and nt = 10. Task cutoff at MIN_CHUNK	Cilk	Tasking
`BFS`	Breadth first search algorithm	Input as graph4M.txt. Chunks = 512 (first parallel for) and Chunks = 4192 (second parallel for)	Rodinia [9]	Parallel for
`Hotspot`	Iterative thermal simulation	grid_rows = grid_cols = 4096, sim_time = 10, temp_file = temp_4096, power_file = power_4096. Chunks = 1	Rodinia	Parallel for
`Srad`	Diffusion method based on partial differential equations	rows = cols = 4096, y1 = x1 = 0, y2 = x2 = 127, lambda = 0.5, iterations = 2. Chunks = 1	Rodinia	Parallel for
`LUD`	Decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix	Matrix dimension = 4096 and block size = 64. Chunks = 1	Rodinia	Parallel for
`B+Tree`	Similar to binary search tree but each node can have up to n − 1 keys instead of just two	file = mil.txt and command = command2.txt. Chunks = 32	Rodinia	Parallel for

by removing all OpenMP pragmas. For all three configurations (`ARM-only`, `DSP-only`, and `Hybrid`) the measurements are reported using all available cores under that configuration, i.e., `ARM-only` uses all 4 ARM cores, `DSP-only` uses all 8 DSP cores, and `Hybrid` uses all 12 cores (4 ARMs and 8 DSPs). Task cutoff in tasking type and total chunks in `parallel for` type benchmarks were chosen such that they achieved the best performance in each of the four parallel versions. A `static` schedule was used in both OpenMP multicore and accelerator model variants of each benchmark as it delivered the best performance. Each of the five implementations was executed ten times and we report the mean of the execution time, along with a 95% confidence interval. To generate ARM binaries, the ARM Linaro gcc compiler version 4.7.3 was used with these flags: `-O3 -mcpu=cortex-a15 -mfpu=vfpv4 -mfloat-abi=hard -fopenmp`. To generate DSP binaries with OpenMP, the TI CLACC OpenMP Accelerator Model Compiler version 1.2.0 was used with these flags: `--hc=‘‘-O3 -fopenmp -marm -mfloat-abi=hard’’ --tc=‘‘-O3’’`. To generate DSP binaries for HC-K2H and HetroOMP the TI C66x compiler *cl6x* version 8.0.3 was used with flags: `abi=eabi -mv6600 -op3 -ma multithread -O3`.

8 Experimental Evaluation

8.1 Total Number of Steals in PvtDeque v/s HardwareQueue

In Sect. 5 we described our choice of PvtDeque based implementation for `DSP_WS` in HetroOMP. It is basically for reducing load imbalance between DSP workers by supporting FIFO steal operations. In this section the benefit of this approach is illustrated. For each benchmark, we calculated the total number of steals during `DSP-only` execution across both HC-K2H and HetroOMP runtimes. The ratio between the result obtained for HC-K2H and that of HetroOMP is then measured. Results of this experiment are shown in Fig. 4. We can observe that the total number of steals among DSP workers in HC-K2H is 76× (`Hotspot`) to 4.5× (`MergeSort`) of that in HetroOMP. The reason for this wide variation is task granularity as there are always lesser number of steals for coarse granular tasks than fine granular tasks. Both HetroOMP and HC-K2H execute a `parallel for` loop in a recursive divide-and-conquer fashion. Hence, for both tasking and `parallel for` type benchmarks, FIFO steals based PvtDeque displace a significant chunk of computation unlike the LIFO steal based HardwareQueue implementation inside HC-K2H.

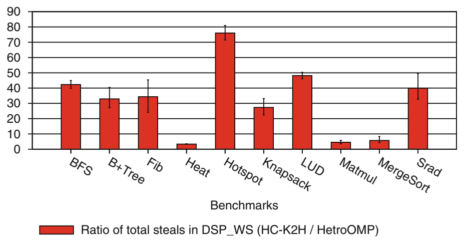
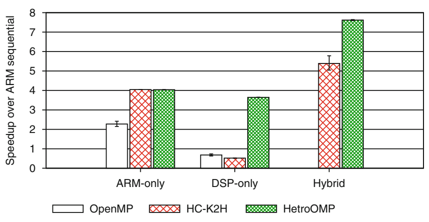


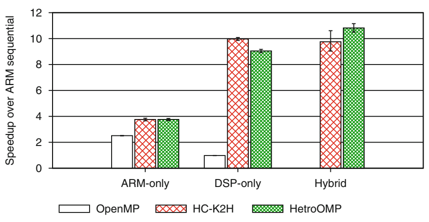
Fig. 4. Total number of `DSP-only` steals in HC-K2H normalized to HetroOMP

8.2 Performance Analysis

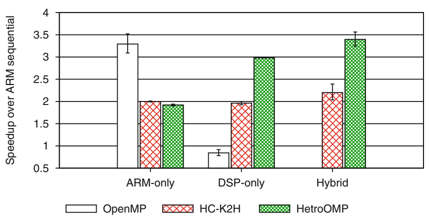
In this section, we describe the performance of HetroOMP on K2H. For this study, all three versions of each benchmark (HetroOMP, HC-K2H, and OpenMP) were executed, first by using all four ARM cores only (`ARM-only`), and then by using all eight DSP cores only (`DSP-only`). Apart from this, hybrid execution



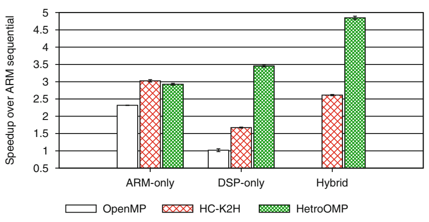
(a) Fib



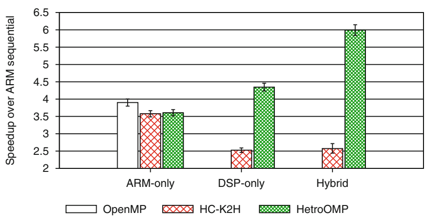
(b) Matmul



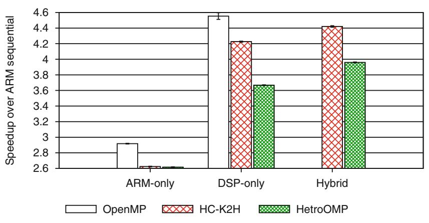
(c) Knapsack



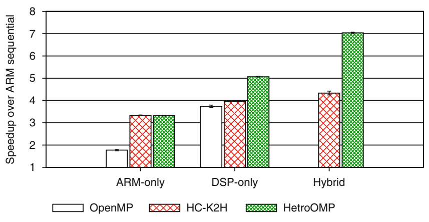
(d) MergeSort



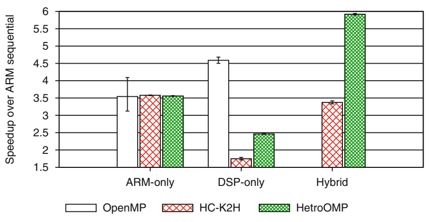
(e) Heat



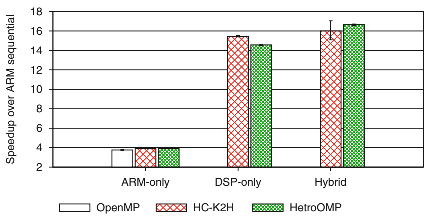
(f) BFS



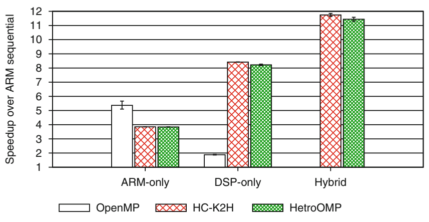
(g) Hotspot



(h) Srad



(i) LUD



(j) B+Tree

Fig. 5. Speedup over sequential execution at ARM. Benchmarks in (a)–(e) are of tasking type whereas those in (f)–(j) are `parallel for` type.

of HetroOMP and HC-K2H implementations was also performed across all four ARM and eight DSP cores (`Hybrid`). The speedup was then calculated for each execution (`ARM-only`, `DSP-only`, and `Hybrid`) over Sequential execution on ARM core. Results of this experiment are shown in Fig. 5. OpenMP's `DSP-only` execution of `Heat` and `LUD` did not complete due to which the results of these experiments are missing in Figs. 5(e) and (i).

Tasks Based Benchmarks: Figures. 5(a)–(e) show the experimental results for tasking benchmarks. These benchmarks recursively spawn and synchronize on asynchronous tasks similar to `MergeSort` implementation shown in Fig. 2(b). We can observe that due to the reduced number of steals in `DSP_WS` and due to the reduced number of CacheWBInv operations, HetroOMP outperformed HC-K2H for both `DSP-only` and `Hybrid` executions. `Matmul` is an outlier as its performance with both these runtimes are in the same ballpark. This is due to high steal ratio in `Matmul`, unlike all other benchmarks. It was found to be around 65% for `DSP-only` and `Hybrid` execution in both these runtimes. `ARM_WS` implementation is similar across HetroOMP and HC-K2H resulting in identical performance. `Hybrid` execution of HetroOMP always outperformed `ARM-only` and `DSP-only` based OpenMP executions (except `Knapsack`).

Parallel for Type Benchmarks: Figures 5(f)–(j) show the experimental results for `parallel for` benchmarks. `ARM-only` execution is again identical across both HetroOMP and HC-K2H (explained above). For `DSP-only` and `Hybrid` executions, HetroOMP performance relative to HC-K2H was in the range 0.86×–1.7× (higher is better). In spite of the benefits of FIFO steals, PvtDeque also has a limitation that it performs slightly weak for coarse granular tasks. It is because the victim is not able to quickly respond to a steal request while executing coarse granular tasks compared to fine granular tasks. Also, due to an implicit barrier at the end of `pragma omp for`, these benchmarks are of flat `finish` type, i.e., a single level of task synchronization. Optimizations for reducing CacheWBInv in HetroOMP are suitable only for nested `finish` type benchmarks and hence are not enabled during the execution of flat `finish` benchmarks. Here too `Hybrid` execution of HetroOMP always outperformed `ARM-only` and `DSP-only` based OpenMP executions. `BFS` is an outlier as even with bigger chunk sizes (see Table 1), both HC-K2H and HetroOMP incurred tasking overheads due to the largest number of tasks (around 120 K while the average number across five benchmarks was 53K). Unlike HC-K2H and HetroOMP, OpenMP executions used `static` schedule where tasks are statically assigned to the threads. Overall, HetroOMP and HC-K2H obtained a geometric mean speedup of 3.6× and 2.6× respectively over `DSP-only` OpenMP execution.

9 Conclusion and Future Work

In this paper, we studied the limitations of the OpenMP accelerator model by using a heterogeneous MPSoC. We demonstrated that for achieving optimal performance, it is essential to utilize the computing power of all the processing elements instead of solely using the accelerator. We proposed extensions to the standard OpenMP accelerator model to enable simultaneous execution across both host and accelerator devices. We presented and evaluated a novel hybrid work-stealing runtime for OpenMP that efficiently executed computation across all processing elements of a heterogeneous SoC and outperformed standard OpenMP accelerator model. As a future work, we aim to extend HetroOMP with energy efficient execution capabilities.

Acknowledgments. We are grateful to the anonymous reviewers for their suggestions on improving the presentation of the paper, and to Eric Stotzer from Texas Instruments for shipping a brand new TI Keystone-II MPSoC to IIIT Delhi.

References

1. Clang LibTooling, April 2019. https://clang.llvm.org/docs/LibTooling.html
2. OpenMP API, version 4.5, March 2018. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
3. Acar, U.A., Chargueraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private deques. In: PPoPP, pp. 219–228 (2013). https://doi.org/10.1145/2442516.2442538
4. Aguilar, M.A., Leupers, R., Ascheid, G., Murillo, L.G.: Automatic parallelization and accelerator offloading for embedded applications on heterogeneous MPSoCs. In: DAC, pp. 49:1–49:6 (2016). https://doi.org/10.1145/2897937.2897991
5. Barik, R., Farooqui, N., Lewis, B.T., Hu, C., Shpeisman, T.: A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In: CGO, pp. 70–81 (2016). https://doi.org/10.1145/2854038.2854052
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46** (1999). https://doi.org/10.1145/324133.324234
7. Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., Gatherer, A.: Implementing OpenMP on a high performance embedded multicore MPSoC. In: IPDPS, pp. 1–8 (2009). https://doi.org/10.1109/IPDPS.2009.5161107
8. Charles, P., Grothoff, C., Saraswat, V., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA, pp. 519–538 (2005). https://doi.org/10.1145/1094811.1094852
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: IISWC, pp. 44–54 (2009). https://doi.org/10.1109/IISWC.2009.5306797
10. 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 Process. Lett. **21**(02), 173–193 (2011). https://doi.org/10.1142/S0129626411000151
11. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI, pp. 212–223 (1998)
12. Grossman, M., Shirako, J., Sarkar, V.: OpenMP as a high-level specification language for parallelism. In: IWOMP, pp. 141–155 (2016). https://doi.org/10.1007/978-3-319-45550-1_11

13. Kumar, V., Sbîrlea, A., Jayaraj, A., Budimlić, Z., Majeti, D., Sarkar, V.: Heterogeneous work-stealing across CPU and DSP cores. In: HPEC, pp. 1–6 (2015). https://doi.org/10.1109/HPEC.2015.7322452
14. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: HabaneroUPC++: a compiler-free PGAS library. In: PGAS 2014 (2014). https://doi.org/10.1145/2676870.2676879
15. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 287–296. ASPLOS (2008). https://doi.org/10.1145/1346281.1346318
16. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO, pp. 45–55 (2009). https://doi.org/10.1145/1669112.1669121
17. Mitra, G., Bohmann, J., Lintault, I., Rendell, A.P.: Development and application of a hybrid programming environment on an ARM/DSP system for high performance computing. In: IPDPS, pp. 286–295 (2018). https://doi.org/10.1109/IPDPS.2018.00038
18. Mitra, G., Stotzer, E., Jayaraj, A., Rendell, A.P.: Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture. In: Using and Improving OpenMP for Devices, Tasks, and More, pp. 202–214 (2014). https://doi.org/10.1007/978-3-319-11454-5_15
19. Munshi, A.: The OpenCL specification. In: IEEE Hot Chips, pp. 1–314 (2009)
20. Compute Unified Device Architecture Programming Guide, April 2019
21. ORNL: Summit supercomputer. https://www.olcf.ornl.gov/summit/. Accessed April 2019
22. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. IJHPCA **23**(3), 284–299 (2009). https://doi.org/10.1177/1094342009106195
23. Sbîrlea, A., Zou, Y., Budimlíc, Z., Cong, J., Sarkar, V.: Mapping a data-flow programming model onto heterogeneous platforms. LCTES **47**, 61–70 (2012). https://doi.org/10.1145/2248418.2248428
24. Texas Instruments: C66AK2H multicore DSP+ARM Keystone II System-On-Chip. Texas Instruments Literature: SPRS866
25. Paderborn University: Noctua supercomputer. https://pc2.uni-paderborn.de/about-pc2/announcements/news-events/article/news/supercomputer-noctua-inaugurated/. Accessed April 2019
26. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC first experiences with real-world applications, pp. 859–870. EuroPar (2012). https://doi.org/10.1007/978-3-642-32820-6_85